The Task and Workslip Scheme for Personal Agent Systems

Kendon, Tyson James


Downloaded from PRISM Repository, University of Calgary
The Task and Workslip Scheme for Personal Agent Systems

by

Tyson James Kendon

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA
MAY, 2019

© Tyson James Kendon 2019
Abstract

The Task and Workslip Scheme for Personal Agent Systems is a collection of concepts enabling the intelligent communication of small agents, and a set of philosophical guidelines to aid in the design of intelligent systems to assist people in day-to-day tasks. The intention behind the scheme is to empower people to take control of their own computing power to achieve the kind of automatic personal assistance that otherwise can only be achieved with corporate computing.

The scheme addresses the issue of how to provide people with these tools in three ways. It provides a theoretical framework for the design of small pieces of software, Agents, which communicate using usefully structured information, Workslips, and can combine to form helpful systems for people, Personal Agent Systems. It provides a set of philosophical guidelines which suggest particular design choices developers should make while developing these Agents. Finally it provides a reference implementation, TWPARI-J, which can be used to develop these Agents and provides practical examples of the philosophical guidelines.

To demonstrate the effectiveness of TWPARI-J and the applicability of the philosophical guidelines and conceptual framework, three example systems are presented which allow people to: adjust their alarm clocks according to the weather, plan their shopping according to sales prices of local stores, and to get assistance monitoring the state of their network in an intelligent fashion. For each system particular instantiations are provided where the system is able to help a particular person.

The scheme provides a simple system to create personal assistant system, often without needing to write new software, and simplifies Agent development. The scheme allows people to use the system and develop applications with a maximum of flexibility using a minimum of work.
Acknowledgements

My thanks first go to Professor Jörg Denzinger, who has been my guide, sounding board, sanity checker, and reader of endless chapters and fixes.

I would also like to thank Professor Jeffrey Boyd for his sage advice and support.

I would like to thank Professors Roberto Flores-Mendes, Patrick Finn and Christian Jacob for their excellent feedback and exciting ideas to continue this research.

The support of my colleagues in teaching both at Mount Royal University and at University of Calgary have been invaluable both for their introduction into the profession of teaching and for their support while I finished this task.

My friends and family have also been incredibly supportive while I found my way to here. Leanne, you have been my rock finishing this and I appreciate your help in finding all the lost commas and more.
Leanne

All the way to the Oort Cloud and back.
Table of Contents

Abstract                                               ii
Acknowledgements                                       iii
Dedication                                              iv
Table of Contents                                       v
List of Figures and Illustrations                      xi
List of Tables                                          xiii
Epigraph                                               xvi
1 Introduction                                          1

2 Motivation: Why Personal Agent Systems Matter and Why People Want Personal Agent Systems 9
  2.1 How To Sleep as Late as Possible and Still Be On Time Despite the Weather and Traffic ......................................................... 13
  2.2 How To Get the Best Possible Price on Groceries Without Driving Around Too Much ................................................................. 17
  2.3 How To Know When Your Systems are Down and What To Do About It The Second Something Happens .................................................. 21
  2.4 What People Want from Personal Agent Systems ................................................. 23

3 The Context of Personal Agent Systems                25
  3.1 Help from Artificial Intelligence .......................................................... 26
  3.2 How Siri and Friends Moved In - Personal Assistants in Everyday Lives .... 28
  3.3 Personal Agent Systems and Personal Agent System Frameworks ............... 30
  3.4 Methods for Agent Engineering and Communication ................................... 49
      3.4.1 BDI-Architecture in general .......................................................... 49
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>FIPA-ACL / FIPA-AIPS</td>
<td>53</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Hive</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>Definitions of the Task and Workslip Scheme for Personal Agent Systems</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Principal</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Task, Task Steps and Task Instances</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Workslips and Master Workslips</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Communication Structure</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Environment</td>
<td>64</td>
</tr>
<tr>
<td>4.6</td>
<td>Agent Definitions and Agent Instances</td>
<td>67</td>
</tr>
<tr>
<td>4.7</td>
<td>Personal Agent Systems</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>Philosophical Guidelines for the Design and Implementation of Personal Agent Systems using the Task and Workslip Scheme</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Tasks That are Well-Suited for the Task and Workslip Scheme</td>
<td>80</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Repeated Tasks</td>
<td>81</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Regular Tasks</td>
<td>83</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Informatic Tasks</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>The Design and Implementation of Agents</td>
<td>85</td>
</tr>
<tr>
<td>5.2.1</td>
<td>As Simple as Possible, but No Simpler</td>
<td>86</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Reusable, Written Once and Used Everywhere</td>
<td>87</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Clear in their requirements and production</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>The Structure of Workslips</td>
<td>89</td>
</tr>
<tr>
<td>5.3.1</td>
<td>A Data Structure Sharable Between Agents</td>
<td>90</td>
</tr>
<tr>
<td>5.3.2</td>
<td>A Data Structure Defined by Its History</td>
<td>90</td>
</tr>
<tr>
<td>5.3.3</td>
<td>A Data Structure That Is Sufficiently Specific and Unique</td>
<td>93</td>
</tr>
<tr>
<td>5.4</td>
<td>The Use of Personal Agent Systems</td>
<td>95</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Personal</td>
<td>96</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Automatic, But Guided</td>
<td>97</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Efficient, But Discreet</td>
<td>99</td>
</tr>
<tr>
<td>6</td>
<td>TWPARI-J: A Reference Implementation for Personal Agent Systems using the Task and Workslip Scheme</td>
<td>102</td>
</tr>
<tr>
<td>6.1</td>
<td>Workslip</td>
<td>106</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Workslip Implementation</td>
<td>109</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Shareable Workslip Format</td>
<td>112</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Workslip Lifecycle</td>
<td>113</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Master Workslip Implementation</td>
<td>114</td>
</tr>
<tr>
<td>6.2</td>
<td>Agents</td>
<td>114</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Agent Configuration</td>
<td>116</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Agent Cell</td>
<td>118</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.3 Agent Implementation</td>
<td>121</td>
</tr>
<tr>
<td>6.3 Infrastructure</td>
<td>122</td>
</tr>
<tr>
<td>6.3.1 Communication</td>
<td>123</td>
</tr>
<tr>
<td>6.3.2 Dispatcher</td>
<td>125</td>
</tr>
<tr>
<td>6.3.3 Scripts</td>
<td>126</td>
</tr>
<tr>
<td>6.4 Developing Agents with TWPARI-J</td>
<td>127</td>
</tr>
<tr>
<td>6.4.1 Planning the Agent</td>
<td>128</td>
</tr>
<tr>
<td>6.4.2 Developing the Agent</td>
<td>131</td>
</tr>
<tr>
<td>6.4.3 Running The Agent</td>
<td>132</td>
</tr>
<tr>
<td>6.5 Personal Agent Systems</td>
<td>133</td>
</tr>
<tr>
<td>7 Three Example Systems Using TWPARI-J</td>
<td>136</td>
</tr>
<tr>
<td>7.1 An Aware Alarm Clock</td>
<td>137</td>
</tr>
<tr>
<td>7.1.1 Requirements</td>
<td>138</td>
</tr>
<tr>
<td>7.1.2 Workslip</td>
<td>139</td>
</tr>
<tr>
<td>7.1.3 Agents</td>
<td>145</td>
</tr>
<tr>
<td>7.1.4 System Application</td>
<td>154</td>
</tr>
<tr>
<td>7.2 An Efficient and Price Aware Shopping List</td>
<td>158</td>
</tr>
<tr>
<td>7.2.1 Requirements</td>
<td>159</td>
</tr>
<tr>
<td>7.2.2 Workslip</td>
<td>160</td>
</tr>
<tr>
<td>7.2.3 Agents</td>
<td>163</td>
</tr>
<tr>
<td>7.2.4 System Application</td>
<td>168</td>
</tr>
<tr>
<td>7.3 An Informative and Proactive System Operator</td>
<td>172</td>
</tr>
<tr>
<td>7.3.1 Requirements</td>
<td>174</td>
</tr>
<tr>
<td>7.3.2 Workslip</td>
<td>174</td>
</tr>
<tr>
<td>7.3.3 Agents</td>
<td>178</td>
</tr>
<tr>
<td>7.3.4 System Application</td>
<td>183</td>
</tr>
<tr>
<td>8 Evaluating TWPARI-J Systems</td>
<td>187</td>
</tr>
<tr>
<td>8.1 An Environmentally Aware Alarm Clock</td>
<td>190</td>
</tr>
<tr>
<td>8.1.1 Setting up the Personal Agent Systems</td>
<td>192</td>
</tr>
<tr>
<td>8.1.2 The Personal Agent Systems in Action</td>
<td>210</td>
</tr>
<tr>
<td>8.2 An Efficient and Price Aware Shopping List</td>
<td>230</td>
</tr>
<tr>
<td>8.2.1 Setting up the Personal Agent Systems</td>
<td>231</td>
</tr>
<tr>
<td>8.2.2 The Personal Agent Systems in Action</td>
<td>242</td>
</tr>
<tr>
<td>8.3 An Informative and Proactive System Operator</td>
<td>252</td>
</tr>
<tr>
<td>8.3.1 Setting up the Personal Agent Systems</td>
<td>253</td>
</tr>
<tr>
<td>8.3.2 The Personal Agent Systems in Action</td>
<td>265</td>
</tr>
<tr>
<td>9 Conclusion and Future Work</td>
<td>273</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

**Bibliography**

**A TWPARI-J: Java Documentation**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Package personalagents</td>
<td>296</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Interface Agent</td>
<td>297</td>
</tr>
<tr>
<td>A.1.2</td>
<td>Interface AgentCore</td>
<td>301</td>
</tr>
<tr>
<td>A.1.3</td>
<td>Class AgentInformation</td>
<td>301</td>
</tr>
<tr>
<td>A.1.4</td>
<td>Class AgentInformationSerializer</td>
<td>307</td>
</tr>
<tr>
<td>A.1.5</td>
<td>Class AgentResources</td>
<td>309</td>
</tr>
<tr>
<td>A.2</td>
<td>Package personalagents.agentutilities</td>
<td>318</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Class FloatExtractor</td>
<td>319</td>
</tr>
<tr>
<td>A.3</td>
<td>Package personalagents.cell</td>
<td>320</td>
</tr>
<tr>
<td>A.3.1</td>
<td>Interface AgentClassFactory</td>
<td>321</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Class AgentCell</td>
<td>322</td>
</tr>
<tr>
<td>A.3.3</td>
<td>Class ReflectiveAgentClassFactory</td>
<td>326</td>
</tr>
<tr>
<td>A.4</td>
<td>Package personalagents.cell.executionmodel</td>
<td>327</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Interface AgentExecutionHandler</td>
<td>328</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Interface AgentExecutionHandlerFactory</td>
<td>330</td>
</tr>
<tr>
<td>A.4.3</td>
<td>Interface RegistrationHandler</td>
<td>331</td>
</tr>
<tr>
<td>A.4.4</td>
<td>Class ContinuousFilePassingExecutionHandler</td>
<td>332</td>
</tr>
<tr>
<td>A.4.5</td>
<td>Class EmptyWorkslipOutputExecutionHandler</td>
<td>334</td>
</tr>
<tr>
<td>A.4.6</td>
<td>Class ExecutionHandlerFactory</td>
<td>337</td>
</tr>
<tr>
<td>A.4.7</td>
<td>Class FileWritingExecutionHandler</td>
<td>339</td>
</tr>
<tr>
<td>A.4.8</td>
<td>Class FileWritingRegistrationHandler</td>
<td>342</td>
</tr>
<tr>
<td>A.4.9</td>
<td>Class OneRunFilePassingExecutionHandler</td>
<td>343</td>
</tr>
<tr>
<td>A.4.10</td>
<td>Class RegistrationExecutionHandler</td>
<td>346</td>
</tr>
<tr>
<td>A.4.11</td>
<td>Class VersionExecutionHandler</td>
<td>347</td>
</tr>
<tr>
<td>A.5</td>
<td>Package personalagents.composableAgent</td>
<td>348</td>
</tr>
<tr>
<td>A.5.1</td>
<td>Interface InputHandler</td>
<td>349</td>
</tr>
<tr>
<td>A.5.2</td>
<td>Interface Processor</td>
<td>349</td>
</tr>
<tr>
<td>A.5.3</td>
<td>Class ComposableAgent</td>
<td>350</td>
</tr>
<tr>
<td>A.6</td>
<td>Package personalagents.configuration</td>
<td>351</td>
</tr>
<tr>
<td>A.6.1</td>
<td>Class ConfigurationSet</td>
<td>351</td>
</tr>
<tr>
<td>A.6.2</td>
<td>Exception MissingConfigurationException</td>
<td>357</td>
</tr>
<tr>
<td>A.7</td>
<td>Package personalagents.dispatcher</td>
<td>358</td>
</tr>
<tr>
<td>A.7.1</td>
<td>Interface RegistrationHandler</td>
<td>360</td>
</tr>
<tr>
<td>A.7.2</td>
<td>Interface RegistrationHandlerFactory</td>
<td>361</td>
</tr>
<tr>
<td>A.7.3</td>
<td>Interface ShowAlive</td>
<td>363</td>
</tr>
<tr>
<td>A.7.4</td>
<td>Interface WorkslipArchive</td>
<td>364</td>
</tr>
</tbody>
</table>
### A.7.5 Interface WorkslipIntakeHandler
- Page 366

### A.7.6 Interface WorkslipIntakeHandlerFactory
- Page 367

### A.7.7 Interface WorkslipOutputHandler
- Page 368

### A.7.8 Interface WorkslipOutputHandlerFactory
- Page 369

### A.7.9 Class DispatcherCell
- Page 371

### A.7.10 Class DispatcherTestRunner
- Page 372

### A.7.11 Class DispatchTesting
- Page 373

### A.7.12 Class FileReadingRegistrationHandler
- Page 374

### A.7.13 Class FileReadingWorkslipIntakeHandler
- Page 376

### A.7.14 Class FileWritingWorkslipOutputHandler
- Page 378

### A.7.15 Class FullFileWorkslipArchive
- Page 380

### A.7.16 Class LightTextWorkslipArchive
- Page 381

### A.7.17 Class ManualRegistrationHandlerFactory
- Page 383

### A.7.18 Class ManualWorkslipIntakeHandlerFactory
- Page 384

### A.7.19 Class ManualWorkslipOutputHandlerFactory
- Page 386

### A.7.20 Class SilentAlive
- Page 387

### A.7.21 Class TickAlive
- Page 389

### A.7.22 Class WorkslipDispatcher
- Page 391

### A.8 Package personalagents.masterworkslip

### A.8.1 Class MasterWorkslip
- Page 396

### A.9 Package personalagents.utilities

### A.9.1 Class RegexMatcher
- Page 401

### A.9.2 Class WorkslipProducerEngine
- Page 404

### A.10 Package personalagents.workslip

### A.10.1 Interface ObjectDataSerializer
- Page 405

### A.10.2 Interface WorkslipInterface
- Page 406

### A.10.3 Class DefaultObjectDataSerializer
- Page 411

### A.10.4 Class Workslip
- Page 413

### A.10.5 Class WorkslipField
- Page 422

### A.10.6 Class WorkslipSerializer
- Page 427

### A.10.7 Class WorkslipWriter
- Page 429

### B System Configuration

### B.1 An Environmentally Aware Alarm Clock
- Page 431

#### B.1.1 System $A_1$ for Owen
- Page 431

#### B.1.2 System $A_2$ for Tala
- Page 435

#### B.1.3 Alarm Clock System $A_3$ for Thad
- Page 439

#### B.1.4 Alarm Clock System $A_4$ for Fiona
- Page 444

### B.2 An Efficient and Price Aware Shopping List
- Page 449

#### B.2.1 System $L_1$ for Olive
- Page 449
TABLE OF CONTENTS

B.2.2 System $L_2$ for Taylor ................................................. 453
B.2.3 System $L_3$ for Thea .................................................... 457
B.3 An Informative and Proactive System Operator ......................... 461
  B.3.1 System $P_1$ for Oscar ................................................. 461
  B.3.2 System $P_2$ for Tress ............................................... 466

C (Selected) Workshop Results ................................................. 473
  C.1 An Environmentally Aware Alarm Clock ............................... 473
    C.1.1 System $A_1$ for Owen ............................................. 473
    C.1.2 System $A_2$ for Tala .............................................. 522
    C.1.3 System $A_3$ for Thad .............................................. 550
    C.1.4 System $A_4$ for Fiona ............................................ 595
  C.2 An Efficient and Price Aware Shopping List .......................... 609
    C.2.1 System $L_1$ for Olive ............................................. 609
    C.2.2 System $L_2$ for Taylor ............................................ 611
    C.2.3 System $L_3$ for Thea .............................................. 613
  C.3 An Informative and Proactive System Operator ...................... 615
    C.3.1 System $P_1$ for Oscar ............................................. 615
    C.3.2 System $P_2$ for Tress ............................................. 618
List of Figures and Illustrations

1.1 The Structure of the Task and Workslip Scheme .......................... 3
6.1 A simple personal agent system ............................................ 105
6.2 A simple agent ............................................................... 106
6.3 Workslip Interface Diagram .................................................. 109
6.4 Workslip Diagram ............................................................. 110
6.5 Workslip Field Diagram ....................................................... 112
6.6 Configuration Overwriting ................................................... 118
7.1 The workslip flow of the Aware Alarm Clock. ............................ 146
7.2 The workslip flow of the Aware Shopping List ............................. 164
7.3 The workslip flow of the Proactive System Operator ....................... 179
8.1 The workslip flow of Owen’s Environment Aware Alarm Clock ......... 193
8.2 The workslip flow of Tala’s Environment Aware Alarm Clock .......... 197
8.3 The workslip flow of Thad’s Environment Aware Alarm Clock .......... 200
8.4 The workslip flow of Fiona’s Environment Aware Alarm Clock ........ 206
8.5 Owen’s System in Action, 1 ................................................. 210
8.6 Owen’s System in Action, 2 ............................................... 211
8.7 Owen’s System in Action, 3 ............................................... 212
8.8 Owen’s System in Action, 4 ............................................... 213
8.9 Data File for the WC\(_1\) weather example .................................. 217
8.10 Owen’s First Workslip, Page 1 ........................................... 218
8.11 Owen’s First Workslip, Page 2 ........................................... 219
8.12 Owen’s First Workslip, Page 3 ........................................... 220
8.13 Owen’s First Workslip, Page 4 ........................................... 221
8.14 Owen’s Second Workslip ................................................... 222
8.15 Tala’s Fourth Workslip ...................................................... 224
8.16 Thad’s Workslip For the Worst Weather ................................... 226
8.17 Fiona’s Normal Rain \(WV_2\) .................................................. 228
8.18 Fiona’s Heavy Rain \(WV_3\) ................................................... 229
8.19 The workslip flow of all Olive, Taylor and Thea’s shopping list personal agent systems. ................................................................. 232
8.20 Olive’s First Shopping Plan .................................................. 244
8.21 Olive’s Second Shopping Plan .............................................. 246
8.22 Olive’s Third Shopping Plan ................................................ 247
8.23 Olive’s Final Shopping Plan ................................................. 248
8.24 Taylor’s Final Shopping Plan ................................................. 250
8.25 Thea’s Final Shopping Plan .................................................... 252
8.26 The workslip flow of Oscar’s Proactive System Operator. .......... 256
8.27 The workslip flow of Tress’s Proactive System Operator. .............. 261
8.28 A workslip from Oscar’s PSO in action .................................. 268
8.29 A later workslip from Oscar’s PSO in action .............................. 269
8.30 A workslip from Tress’s PSO in action ..................................... 271
List of Tables

7.2 Field templates in the Aware Alarm Clock for the weather evaluation . . . . 143
7.1 Fields in the Aware Alarm Clock for the weather report, types and potential
values of the field are based on information from [Env14]. . . . . . . . . . . . 143
7.3 The field template in the Aware Alarm Clock for the alarm update . . . . . 144
7.4 The identity information for an Agent . . . . . . . . . . . . . . . . . . . . . . 145
7.5 The identity information for the Weather Source Agent . . . . . . . . . . . . 148
7.6 The identity information for the Weather Evaluation Agent . . . . . . . . . . 149
7.7 The identity information for the Weather Evaluation Agent . . . . . . . . . . 151
7.8 The alarm update table for an example alarm for a principal . . . . . . . . . . 152
7.9 The identity information for the Weather Evaluation Agent . . . . . . . . . . 153
7.10 The field template in the Aware Shopping List for the principal’s shopping list 161
7.11 The items available from the store \( ⟨store⟩ \) . . . . . . . . . . . . . . . . . . . 162
7.12 The items available from the store \( ⟨store⟩ \) . . . . . . . . . . . . . . . . . . . 162
7.13 The identity information for the Shopping List Agent . . . . . . . . . . . . . 164
7.14 The identity information for the Store Information Agent . . . . . . . . . . . 165
7.15 The identity information for the Store Information Agent . . . . . . . . . . . 166
7.16 The identity information for the Principal Communication Agent . . . . . . 167
7.17 The field template for the Proactive System Operator (PSO) for the system
status fields. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
7.18 The field template for the Proactive System Operator for the system reaction
fields. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
7.19 The field template for the Proactive System Operator for the system update
fields. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177
7.20 The identity information for a Service Testing Agent . . . . . . . . . . . . . 180
7.21 The identity information for a Service Reaction Agent . . . . . . . . . . . . . 181
7.22 The identity information for the System Status Agent . . . . . . . . . . . . . 182
7.23 The identity information for the Principal Communication Agent . . . . . . 183

8.1 Four artificial principals and the task instances that they would like to fulfill. 191
8.2 Owen’s Configuration for the Weather Source Agent, \( WSA \) . . . . . . . . . . 193
8.3 Owen’s Configuration for the Weather Evaluation Agent, \( WEA \) . . . . . . . . 194
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4</td>
<td>Owen’s Configuration for the Alarm Control Agent, ACA</td>
<td>195</td>
</tr>
<tr>
<td>8.5</td>
<td>Owen’s Configuration for the Alarm Control Agent, ASA</td>
<td>196</td>
</tr>
<tr>
<td>8.6</td>
<td>Tala’s Configuration of the Weather Evaluation Agent, WEA</td>
<td>198</td>
</tr>
<tr>
<td>8.7</td>
<td>Tala’s Configuration for the Alarm Control Agent, ACA</td>
<td>199</td>
</tr>
<tr>
<td>8.8</td>
<td>Configuration for Tala’s Alarm Sounding Agent, ASA_2</td>
<td>199</td>
</tr>
<tr>
<td>8.9</td>
<td>Thad’s Configuration for Forecast Parser Agent, FPA</td>
<td>201</td>
</tr>
<tr>
<td>8.10</td>
<td>Thad’s a Configuration for the instance of the Weather Evaluation Agent, WEA^a</td>
<td>202</td>
</tr>
<tr>
<td>8.11</td>
<td>Thad’s b Configuration for the instance of the Weather Evaluation Agent, WEA^b</td>
<td>203</td>
</tr>
<tr>
<td>8.12</td>
<td>Thad’s Configuration for the Alarm Control Agent, ACA</td>
<td>204</td>
</tr>
<tr>
<td>8.13</td>
<td>Configuration for Thad’s Alarm Sounding Agent, ASA_3</td>
<td>205</td>
</tr>
<tr>
<td>8.14</td>
<td>Fiona’s Weather Configuration for Source Agent, WSA</td>
<td>206</td>
</tr>
<tr>
<td>8.15</td>
<td>Fiona’s Configuration for the Forecast Parser Agent, FPA</td>
<td>207</td>
</tr>
<tr>
<td>8.16</td>
<td>Fiona’s Configuration for the Weather Evaluation Agent, WEA</td>
<td>208</td>
</tr>
<tr>
<td>8.17</td>
<td>Configuration for Fiona’s Alarm Control Agent, ACA</td>
<td>208</td>
</tr>
<tr>
<td>8.18</td>
<td>Configuration for Fiona’s Alarm Sounding Agent, ASA</td>
<td>209</td>
</tr>
<tr>
<td>8.19</td>
<td>Three artificial principals and the task instances of the Price Aware Shopping List that they would like to fulfill.</td>
<td>231</td>
</tr>
<tr>
<td>8.20</td>
<td>The shared configuration for the Shopping List Agent, SLA</td>
<td>233</td>
</tr>
<tr>
<td>8.21</td>
<td>The shopping list shared by Olive, Taylor and Thea</td>
<td>233</td>
</tr>
<tr>
<td>8.22</td>
<td>The shared configuration for the Store Information Agent A, SIA_A</td>
<td>234</td>
</tr>
<tr>
<td>8.23</td>
<td>The shared configuration for the Store Information Agent B, SIA_B</td>
<td>234</td>
</tr>
<tr>
<td>8.24</td>
<td>The shared configuration for the Store Information Agent C, SIA_C</td>
<td>235</td>
</tr>
<tr>
<td>8.25</td>
<td>The shared store information for Store A, including the Item, Brand and price per unit (PPU)</td>
<td>235</td>
</tr>
<tr>
<td>8.26</td>
<td>The shared store information for Store B, including the Item, Brand and price per unit (PPU)</td>
<td>236</td>
</tr>
<tr>
<td>8.27</td>
<td>The shared store information for Store C, including the Item, Brand and price per unit (PPU)</td>
<td>236</td>
</tr>
<tr>
<td>8.28</td>
<td>Configuration for Olive’s Distance Minimizing Shopping Plan Agent.</td>
<td>237</td>
</tr>
<tr>
<td>8.29</td>
<td>Configuration for Olive’s Principal Communication Agent.</td>
<td>238</td>
</tr>
<tr>
<td>8.30</td>
<td>Configuration for Taylor’s Cost Minimizing Shopping Plan Agent.</td>
<td>239</td>
</tr>
<tr>
<td>8.31</td>
<td>Configuration for Taylor’s Principal Communication Agent.</td>
<td>240</td>
</tr>
<tr>
<td>8.32</td>
<td>Configuration for Thea’s Brand Prioritizing Shopping Plan Agent.</td>
<td>241</td>
</tr>
<tr>
<td>8.33</td>
<td>Configuration for Thea’s Principal Communication Agent.</td>
<td>242</td>
</tr>
<tr>
<td>8.34</td>
<td>Two artificial principals demonstrating different use cases of the Proactive System Operator.</td>
<td>253</td>
</tr>
<tr>
<td>8.35</td>
<td>The shared configuration for the Service Testing Agent for the HTTP Service, STA_HTTP</td>
<td>254</td>
</tr>
</tbody>
</table>
8.36 The shared configuration for the Service Testing Agent for the SSH Service, $STA_{SSH}$ ........................................................... 254
8.37 Oscar’s configuration for his System Status Agent ..................................... 258
8.38 The configuration for Oscar’s Ultra Priority Logging PCA ........................... 259
8.39 The configuration for Oscar’s High Priority Logging PCA ......................... 259
8.40 The configuration for Oscar’s Regular Priority Logging PCA .................... 260
8.41 The configuration for Oscar’s Low Priority Logging PCA ............................ 260
8.42 The configuration for Tress’s $SRA_{HTTP}$ .................................................. 262
8.43 The configuration for Tress’s $SRA_{SSH}$ ..................................................... 262
8.44 Tress’s configuration for her System Status Agent ...................................... 263
8.45 The configuration for Tress’s High Priority Logging PCA ............................ 264
8.46 The configuration for Tress’s Regular Priority Logging PCA ....................... 264
8.47 The configuration for Tress’s Low Priority Logging PCA ............................ 265
Epigraph

Oh, I get by with a little help from my friends.

- The Beatles, With a Little Help from My Friends
Chapter 1

Introduction

The Task and Workslip Scheme for Personal Agent Systems allows people to assign important tasks to Personal Agent Systems, without being forced to sacrifice their personal autonomy or share their private information. The scheme consists of a set of definitions, describing how systems can be constructed, a set of philosophical guidelines, describing how systems should be constructed, and a reference implementation showing how systems can be constructed.

The motivation behind the scheme is the freedom a person can have when hiring a person to do a task for them. They are able to have the task done in exactly the way they want it done, without having to do all of the work themselves. Academic research into the field of personal agent systems has been long standing, but is often largely focused on using the systems as test-beds for Artificial Intelligence and Machine Learning techniques, and the systems often are only adequate at best for supporting people. Commercial “Virtual Assistants” (such as Apple’s Siri, Amazon’s Assistant, Google’s Assistant, and many others) afford practical assistance for people, but at a cost of exposing personal information to third parties (at a minimum the party providing the assistant) and a cost of losing personal autonomy in deciding how these assistants will do a task.
While both types of personal agent system are sufficiently effective (and improvements in their effectiveness will stem from improvements generally in computing and specifically in Artificial Intelligence), both suffer from flaws that limit their ability to help people do tasks. The first such flaw is the reliance on complex agents, which use AI or ML techniques to guess how the person would like the job done. While this may be able to help in certain situations where the person does not know how the task is to be done, generally if the person already understands the task, then the best result these approaches can provide is arriving at the same understanding the person already has. This can be circumvented by providing ecosystems of agents, which the person can adopt to do a task that is already understood, thus saving the time spent learning a solution to an already solved problem.

The second major flaw is the profit motive in providing these systems. Generally all Virtual Assistants are provided to people for free (or incorporated in the cost of their operating system). The costs to operate these systems then are deferred to the provider, which will look to advertising revenue, or data mining as models to fund the existence of the assistants. Even in the best-case scenario of a provider who only wishes to garner good will, the decisions about how these systems will work (and what resources they will use) are not made by the people using the systems. Allowing people to use their own computing resources to work on tasks the way they would like them worked on, by creating Personal Agent Systems out of agents, purchased from an open, public source, protects them from these outside parties. This allows them the autonomy to decide how they would like their tasks carried out and as an extension which personal information they are willing to share.

The Task and Workslip Scheme addresses these two flaws by addressing the question: how should a system of modular software be designed so that it allows people to experience the benefits of automation for difficult, and personal, problems, without requiring them to expose their personal information with third parties.
The scheme answers this question of how to build these personal agent systems in three ways: first it provides a theoretical framework for the design of small pieces of software, Agents, which communicate using structured information, Workslips, and can combine to form helpful systems for people, who have tasks to do, so called Principals. Secondly, it provides a set of philosophical guidelines which suggest particular design choices developers should make while developing these Agents. It provides a reference implementation which can be used to simplify the development of these Agents and provides practical examples of the definitions and philosophical guidelines, which are illustrated with three examples systems (as seen in Figure 1.1).

The Task and Workslip Scheme is related to other modular software systems in that it suggests that the output of one program should be the input to others. In this way, it follows
the Unix Philosophy[Ray03], of small pieces of software, each expert at one activity and able to communicate via a clear, and clean interface. The flexibility the Unix philosophy provides to system administrators (and users) to combine programs to create powerful programs, or produce expressive outputs is key. The Task and Workslip Scheme provides the same kind of flexibility, in its agents which can be composed into personal agent systems, in particular due to added information that can be shared through the workslips.

The types of problems for which the Task and Workslip Scheme is intended to solve are tasks which can be broken down into several steps that the person doing the task must do. These tasks must also be able to be done from a computing environment. The dependence on computing makes it simpler to design personal agent systems to work with them, and the number of steps (or the size of the task) must be sufficient to make creating the personal agent system worthwhile.

Consider a webcomic artist who posts a note about their new comic being available to several different channels. By hand, the artist would need to release the comic and then copy the link, and any other information into each of their channels. The Task and Workslip Scheme would allow them to simplify the work. The artist could have one Agent which is responsible for releasing the comic, which would then fill out a workslip with the information necessary about the comic just released, including the link and possibly even an image of the comic itself. That workslip can then be passed to other agents, such as one for Twitter, or one for Facebook, which could then automatically take the workslip with its information and post the information in the correct form and style for each channel.

The workslip is powerful because of the structured nature of the information it contains. Each workslip contains a set of fields which agents can read and update. Each field has a common name, a type, a set of potential values, and a human readable description. When agents work together to form a personal agent system, they begin by announcing which
fields they are interested in reading, and which fields they are able to update. After that introduction, agents can send worksips to all other agents that are interested in the work they just did.

The webcomic artist example is simple, in that it includes only one ‘layer’ of agents. The workslip is produced by the releasing agent, and then the other agents take information from that first update. Other arrangements are also possible. In some cases an agent may be needed to parse, interpret, or otherwise digest information into a form other agents can use. In other cases, agents may need to be invoked only in certain circumstances, and so other agents need to be able to conditionally respond to information. In other systems there may be many updates to one initial workslip and an agent may be needed to compile and recombine those results.

A more complex example might be of a system administrator for a small organization. The organization might have a collection of services (such as a web server, a custom server program, a file server, a database server) that they need for their staff and customers, and the system administrator might have more responsibilities than simply keeping the services up and running. The administrator’s personal agent system could consist of a set of agents which help her to monitor the services, and possibly even to keep them running while she focuses her effort somewhere else.

The administrator could have one set of agents which monitor the services the organization provides. When those services change, for example stop working or experience an error, these agents can take note of that (possibly by poling the service or some other technique), and can update and send a workslip out to other agents.

Depending on the services different responses might be appropriate. If nothing can be done about a service being down then it is simply important that the agent makes the administrator aware of the problem. In other cases, for example a public facing web service,
it might be important to update a health service to indicate that the service has stopped working and that someone is aware and working to solve it, and another agent can be assigned to post this alert. In other cases, for example a custom application with a memory leak, a service might need to be restarted, either on a regular basis or when the service crashes, and stops working. An agent can be assigned to watch for updates from a monitoring agent, and when those updates are received, restart the service. Other agents can be assigned to monitoring, and tracking the overall state of the services and still other agents might be assigned to customize how information from the other agents in the system is digested and communicated with the administrator.

If several webcomic artists were to address the same problem of posting a note about their work being released, each may have a different set of channels they would like it released to, different styles of how they would like the announcement made and different times they might want to send out reminders. The Task and Workslip Scheme accommodates this by allowing them to compose the agents that serve the channels they prefer and configure each in the way they would like the agents to work.

Similarly, different system administrators will have different services they want to monitor. In these situations they will be able to create personal agent systems using different agents that monitor their distinct set of services. They will also have different ways they would like the other agents to work together to communicate with them, and to proactively handle any tasks that can be taken without the administrator.

The Task and Workslip Scheme is intended to provide support and automation to principals without the need to create new software or scripts. The scheme does however rely on some of the problem solving skills that are core to Computer Science and result in algorithmic thinking and solutions. Care must be taken to ensure that while a principal has to do some planning and understand their own task, that this does not overwhelm the principal
and turn the scheme for a personal agent system scheme into a ‘plug and play’ programming language. Instead, agents should be designed in such a way that it is simple for the principal to understand and use and support needs to be provided to make the work of connecting agents as simple as possible.

At the same time, helping principals to understand and take advantage of the benefits this type of problem solving gives, will hopefully help ensure that people interacting with the myriad of different computer systems in the world are better equipped to protect their privacy and autonomy. In the ideal case this will help to ensure in the long term that personal agent systems, particularly ones built using the scheme, will be able to interact with the larger computing world with as few barriers as possible.

The remainder of this thesis is organized as follows. Chapter 2 contains three example scenarios as motivation for the design, and use of the Task and Workslip Scheme. Chapter 3 contains a discussion of the history of personal agent systems, and, more generally, how Artificial Intelligence systems have been designed to assist people with tasks. Chapter 4 contains the first primary contribution, a description of the fundamental concepts needed to define the scheme and personal agent systems designed using the scheme, and Chapter 5 contains the second primary contribution, a description of the philosophical guidelines intended to help people design and develop useful personal agent systems.

The remainder of the primary contribution, is described and then explored in the following chapters. Chapter 6 contains a description of the reference implementation of the scheme as TWPARI-J, which provides numerous tools, constructs and algorithms to enable people to construct personal agent systems. Chapter 7 contains a description of three personal agent systems constructed using TWPARI-J, each of which solves a problem presented in Chapter 2, and Chapter 8 contains an evaluation of the systems in Chapter 7 and illustrates how the personal agent systems function. Finally, Chapter 9 contains a discussion of the future work
to be done with the scheme, and provides a conclusion to the discussion of the scheme.
Chapter 2

Motivation: Why Personal Agent Systems Matter and Why People Want Personal Agent Systems

Why would a person want a personal agent system? People want a system that is particular to them (is personal), but that can get things done (the way they want) without them having to do every step themselves (has agency). Personal agent systems provide this control and flexibility.

Traditionally, software has not been thought of as something personal. This makes sense in light of the history of computation. From the age of Pascal on[Sha95], people wanted computers to calculate the solutions to difficult equations, particularly the ones that human computers[Ski06] found difficult. Much of the origin of modern computing was based on an expectation that software was just filling in blanks in a table of numbers.

Software is mostly written to accomplish a task. The notion of the person using that software is, at best, based on a stereotype. More often than not, software developers write
software based on their own experience and understanding. This results in software which shows the biases of the developers. In some situations, this can result in software that deals with people of different origins very differently, such as in the case of facial recognition software [Tow17]. In other situations, the effect of the software may not be so drastic, but it still produces software that is not a perfect fit for the needs of a person.

When people using software have the same experiences, understanding, and expectations as the software’s developers, the software seems like a good fit for the task the people would like to do. As those experiences, understandings, and expectations diverge, the software becomes an increasingly poor fit for those people.

Some software is also more personal than others. Software that runs a power plant has very little to do with the people who use it. Developers bicker over whether it is better to use VIM, EMACS or an IDE when writing code. This suggests that each one conforms better to a developer’s personal experience, expectation and style, and therefore it must be more personal than the power plant software. Still, developers are able to adapt and use any of these tools to create software.

Software becomes intensely personal as it enters the home. Software is increasingly entering the home as home devices become “smart”. The refrigerator has an interpretation of the level of horseradish stored in it. The algorithms of the “smart refrigerator” affect the lives to the people using it. The horseradish interpretation may not be of concern to some people, but the amount of medicine kept in the fridge might be.

Similarly how lightbulbs react to the time of day can now be a personal concern. Smart devices and the Internet of Things [Wei99] provide dramatic new benefits for people, but they require interacting with software in ways that were never a concern before. This new interaction is tied directly to peoples experiences, understanding and expectations.

Personal agent systems exist to help address these times and places where software be-
comes personal. In particular, the other aspect of personal agent systems, the *agency*, allows people’s wishes for the nature and behaviour of their software to be encoded and enacted.

Agency, as it applies to personal agent systems, reflects two dictionary definitions of the word\(^1\). The agents working within a personal agent system act on behalf of the person running the system (the *principal*). Equally the agents have agency, and are able to take action to produce a particular effect in the world.

Later parts of this document discuss the particulars of designing and developing agents for personal agent systems. Before that, a question should be asked, How do people expect something with agency to work for them? Having agents is not at all new to the human experience, and as such, there is a well established framework that personal agents are expected to fit.

Generally, an agent is a person contracted to do something for another person. This can often involve something that an average person would not be able to do themselves, for example a real estate agent is an expert in both the laws of real-estate and in real-estate negotiation. This can also involve something that a person does not want to do themselves, for example a house-cleaner. In both of these cases the agent acts on behalf of the person who contracted them.

In the case of the real-estate agent, where the person cannot reasonably determine if the agent is acting in their best interest or not, professional organizations exist to certify the behaviour of the agent. In the case of the cleaner it is simpler for the principal to determine if the cleaner acted the way they were supposed to. In either case, at the end of each action, the person can decide if the agent solved the problem in the way they wanted and, if not, choose a different agent, or instruct their agent differently for the next time.

---

\(^1\)From the Oxford English Dictionary, “I. A person or organization acting on behalf of another, or providing a particular service” and “II. Ability or capacity to act or exert power; active working or operation; action, activity.”[Pre19]
The agreement between the person and their agent (as humans) is predicated on money. Agents who do more specialized tasks demand more and prices rise as the work the agent is expected to do becomes more complex or personal. As such having an agent is either a thing done for a very limited time or a thing limited by the wealth of the principal.

A personal assistant, as imagined by popular culture at least, is the epitome of what a personal agent (or personal agent system) would be. A person, hired by a principal, to do things for them in many aspects of their lives. The popular picture of a personal assistant is of someone who keeps the principal’s schedule, who makes sure that the principal’s house is organized, and who makes sure that the principal is able to be in the right places at the right time with the right tools to do what they need to do.

The personal assistant might do these things directly, or they might work as a coordinator. They might hire the cleaner or the driver, rather than clean or drive themselves. They would hire the caterer and set the menu, rather than be in the kitchen. When setting the menu they have the ability to know what the principal would like and then only need agreement or a minor amendment. In each of these cases the personal assistant is able to act on wishes and preferences of their principal.

Having a personal assistant is an expense that does not make sense for most people, unless it is important to the person’s work or the person is wealthy enough that it seems like a reasonable expense. The intention for most personal agent systems is to allow software to recreate, at least some of, the benefits of having a personal assistant at a much lower cost.

To look at the interaction between a principal, a helper (either a personal assistant or a personal agent system) and a task, the rest of this chapter details three scenarios where a person might want some help. Each of these three scenarios can be (and have been) handled by personal agent systems using the Task and Workslip Scheme, and are used to illustrate concepts throughout this document.
In each of these scenarios, Alice, the principal, is a programmer who has just come into a windfall of money. Rather than escape the work-a-day world, Alice decides to see if she can use the money to be an even better, smarter, more organized programmer. She opts to hire a personal assistant, so that she can be as efficient as a Hollywood actor, but directed towards her normal work.

In the first scenario, Alice wants to be able to sleep as late as possible while still getting where she needs to be on time, regardless of weather or traffic. In the second scenario, she wants to buy her groceries at the best price possible depending on which local stores have good sales this week. In the third scenario, she wants to keep track of her network systems, and find and react to failures as quickly as possible.

2.1 How To Sleep as Late as Possible and Still Be On Time Despite the Weather and Traffic

One of the problems that Alice faces on a regular basis is that she lives in Canada, and the weather in Canada can be extremely unpredictable. This means that sometimes, when she goes to bed, she discovers by the morning that the weather is a lot worse then it was forecasted to be. Other times she sets her alarm early to make sure she arrives on time, and the weather turns out to be fine.

Alice is not a natural morning person and she finds being awake earlier than she needs to be quite frustrating. She decides to hire Bob as her personal assistant to help her manage this problem. Bob should monitor the weather overnight and then decide when she needs to wake up to make sure she makes it to her first meeting of the day on time.

While Alice sleeps, Bob brews a pot of coffee and tunes the radio to the oldies music
stations. Bob figures that the announcer will cut in with news about the weather from time to
time and that they will make a special announcement if something particularly bad is going
to happen. The night passes uneventfully, although Bob notices that the announcer does
not mention the weather often. Around eight o’clock in the morning, Bob has the realization
that he does not know when Alice wants to wake up. He wakes Alice immediately, but Alice
is somewhat upset since she has a meeting at eight-thirty and is going to be late.

Later that day Alice and Bob meet. Alice realizes that she is going to need to share
information with Bob in order to make this system work. She takes a few minutes and sets
up her calendar so that Bob can see her first meeting of the day. She also tells Bob that she
would like to wake up exactly two hours before the first meeting so that she can get ready
and drive to work on time.

Alice goes to sleep, again, and Bob, brews another pot of coffee and monitors the radio.
The next morning he wakes Alice up two hours before her first meeting and she is able to
make it to the office on schedule. The next few days go the same way, and Alice feels like
having a personal assistant is paying off.

Later that week, Alice wakes up to discover that it is almost noon. She asks Bob why he
did not wake her up, and he shows her his view of her calendar. Alice had set this day aside
for work and did not have any meetings scheduled at all. Bob followed his instructions and
did not wake her. Alice decides to give Bob a general rule, that she needs to be at the office
at 9am, unless she has an earlier meeting or there is some other exceptional circumstance
that she will tell him about. Additionally she tells Bob that she does not want to be woken
up on Saturday and Sunday mornings.

With basic rules set up things start working fairly well. Alice and Bob go for a few weeks
with the basic system working. Sometimes the radio announcer mentions rain and so Bob
is able to adjust Alice’s wake up time to be early enough. Over time, however, Alice starts
2.1. HOW TO BE ON TIME DESPITE THE WEATHER AND TRAFFIC

to notice that Bob’s accuracy is not as good as she would like. Sometimes Bob wakes her up when the weather does not actually warrant it, and sometimes he does not wake her up early, even when the weather did warrant it.

Alice and Bob sit down to discuss this and Alice asks Bob to explain the whole process he goes through when he decides to change her wake up time. Bob explains about his system of listening for updates from the radio announcer and then, in his experience, any rain would only delay traffic by about 15 minutes, so he wakes her up that much earlier.

Alice is surprised to hear that Bob is using the oldies station since they are broadcast from hundreds of miles away. This means that the forecast Bob is using does not actually help Alice get up on time or not. Bob’s accuracy rate thus far has mostly been luck. Alice asks Bob to use the feed for the government forecast for their city (her preferred forecast). Now Bob has an accurate forecast, and more importantly, a forecast that Alice likes, and Bob’s efficiency greatly improves.

As the season moves from summer into fall and then winter, Bob starts to find that it gets harder to estimate how much different types of weather affect how long it will take Alice to get to work. He asks her to let him know exactly when she makes it to work and he starts to keep track of how long it was for a given weather condition. Over time he starts to build a table of times, which he uses to improve how well he adjusts Alice’s wake up time.

With the weather source settled, rules in place and Bob learning how to estimate the correct amount of time to adjust her wake up, Alice is pleased. Generally Alice is getting to work on time, and problems only seem to crop up when there is something wrong with the traffic, often because of an accident, but sometimes because of planned construction or something else that makes the news such as a festival or a protest.

Alice decides that since she is having Bob stay up all night already, she may as well add to his responsibilities to further smooth her commute. She asks him to start reading the
news to figure out about any other impediments to her drive and to let her know before she
goes to bed what she should expect in the morning. She also asks him to monitor the local
radio station that reports on traffic each morning to help her decide if she should take an
alternate route. She also asks Bob to start including news and traffic information into his
system for deciding when to wake her.

With traffic and news added to the weather, Alice is again pleased with the result. For
the next few months, she gets more sleep on a regular basis and does not have to worry
about arriving late to work. The more she thinks about it however she realizes that she
might not be making the best use of her money. To work out what Bob should do and how
he should do it, she realizes that she had to make several things about her life explicit. This
included things like having to define her preferred source for weather information, having to
define when she expected to get up on days without meetings, having to track how long it
takes her to get to work and where she prefers to get her news and traffic information.

Alice realizes that this explicit information would make it possible for her to write a
program that did all of the work Bob was doing. Alice organized all of the information into
a useful format, such as getting a good weather and traffic report and stripping out the
information that she really cared about. Then she wrote an engine to process all of that
information along with the rules she had and Bob’s adjustment table. The engine was able
to output an adjustment time that she was able to feed to her alarm clock.

Alice tested the system alongside of Bob for a while. The system was not quite as good
as Bob, but was good enough to keep Alice getting to work on time.

Feeling like she has settled this question of getting to work on time, Alice starts looking
for other jobs for Bob to do. Despite her windfall, Alice still likes to be careful managing her
spending on groceries, and she starts to think that it might be time for Bob to start reading
through the fliers, after all the coffee budget seems to have risen steeply this year.
2.2 How To Get the Best Possible Price on Groceries Without Driving Around Too Much

Alice is a creature of habit. Very often when she is shopping for groceries she ends up shopping for the same things every week. She happens to live conveniently near to several grocery stores, and, not having a particular preference for one store over another, tends to pick whichever store seems most convenient on any particular day to go shopping. Alice has noticed that despite her grocery list being pretty similar from week to week, the grocery bill does seem to fluctuate quite a bit.

Up until now Alice has felt that it was not worth spending that much time reading flyers, or going from store to store to make sure she is getting the best possible price on each of the things she buys. Now with Bob finally awake during the day, Alice starts to think that a personal assistant might be the way to make sure she is getting the most value out of her shopping.

Alice decides that the best place to begin is with finding the right information. She sends Bob out to get flyers for all the different stores. Once he has his stack of flyers, she gives him her shopping list, and tells him to figure out where the cheapest place is to buy each of the things on her list.

Bob starts to compile a set of lists for each store, but quickly realizes that he has a problem. The flyers do not provide a complete picture, as items that are not on sale are not included in the flyer. Bob sets out to solve the first problem by asking Alice for all her recent grocery receipts. For each item she has purchased recently he builds up an average price for each item at each store. Now, he can figure out a default place to purchase each thing from the list, and can tell when one store is offering a better deal. For example, he can see that Store A usually sells strawberry jam for $2.77, Store B for $2.90, and Store C for $4.21. It
is not worth looking at a sale price for strawberry jam that does not drop below $2.77. He can also see which stores never carry certain items.

After several weeks of shopping with Bob’s new lists, Alice finds that her grocery bill does start to drop and also to stabilize, even if her time shopping does go up some. Now, she regularly purchases items at the store which offers them for the best price. That being said, Alice notices that overall she is not saving that much money and that a lot of items she used to purchase once a month are now showing up on her list once a week.

She goes back to Bob to see if they can diagnose the problem. Bob shows her his list, and how he keeps an average for each store based on the item being sold. After some checking, Alice realizes that the items Bob is tracking are actually different sizes. Strawberry jam is $2.77 and $2.90 for 250 mL in Store A and Store B, but $4.21 for 500 mL in Store C, making it cheaper than Bob’s “best price”.

Once the flaw in Bob’s original approach comes to light, it works out to be fairly simple for him to implement a fix. For each item he calculates a common “price per X”, for each of the items Alice buys. This takes a little bit of time, because he has to determine what a correct X is for each of the items (100 millilitres of strawberry jam, litres of milk, kilograms of coffee, meters of paper towel and so on). Once that step has been completed through, Bob is able to create new averages for all of the items of Alice’s shopping list based on a common unit.

The new system begins to work fairly well. Alice’s grocery bill drops again, and her shopping time is improved, somewhat, because items appear on the list less frequently than they had before. Alice does notice that despite some improvement her grocery shopping time is still quite long compared to when she went shopping before.

Alice realizes that the problem with this system is that Bob only puts an item on the list if it is strictly cheaper than any of the alternatives. She does some calculations and
realizes that in time spent shopping, and fuel, she does not save money if she only saves a few pennies on the item itself, and she takes longer to shop. To address this, she and Bob add a threshold value for each item. If a store has an item at a cost less than the best price plus the threshold then she will buy the item at that store rather than at the very best price, if the best price means too much driving.

With the threshold system introduced, Alice’s shopping time does drop but she still feels like she is not making the most out of her shopping time. She goes through the process with Bob again, and realizes that the list is specific to brands of items she was buying. Bob always ensured that the brand of the item was preserved, so Alice’s choice of Store X’s brand of dishwashing detergent meant he was only putting the dishwashing detergent on the list for when Alice went to Store A.

Alice tells Bob that she does not care about the brand of dishwashing detergent, and they realize that she has several “I don’t care where they’re from” items. Bob thinks about it, and then, modifies his averages so that, now, he tracks all brands of items for items that Alice does not mind which particular brand of item she purchases and he tracks specific items for the items which Alice does care about.

The new system works fairly well for a while, and Alice is happy that both her shopping time dropped and that the bill has dropped as well. Unfortunately, Alice starts getting some items that she really does not like. For instance, she discovers that Store Z’s brand of dishwashing detergent destroys the finish on her dishes. She also finds, that, while Store B’s brand of Strawberry Jam is acceptable, she prefers Store A’s, and she finds that the milk she buys at Store A spoils much earlier than milk from the other two stores.

Alice and Bob realize that looking at all brands for all items will not work as a general solution. Some items Alice feels are equivalent and for other items she cares about the brand. To fix this, Bob gives Alice a list of all the items he currently tracks, and gets her to note
2.2. HOW TO GET THE BEST PRICE ON GROCERIES

for each of the items, which she considers equivalent. For some items Alice decides that she is happy with any brand, for others she decides any brand is fine so long as it excludes some specific brands, for some she decides that a category can only include a couple of specific brands that she does like, and, for a few, that only the one specific brand will do.

It takes a while for Alice to update all of the information for Bob, but once she does she feels like she really has control over her list, and is still getting the benefits of having Bob manage the lists and find the best prices. She has to update the list of brands from time to time, as her list changes and as the stores’ stock changes, but she finds that generally, the extra work she has to put in is more than outweighed by the benefit she gets.

Again, Alice starts to think that she should be able to write a program to replicate the work Bob does for her. The work Bob is doing is fairly straightforward, and the really difficult part proves to be getting good information from the stores. Once she has that information, she builds a system to replicate both Bob’s system for finding the lowest cost (within a threshold), and the system they developed to determine which items are equivalent. After a little work, she and Bob are convinced that the system is able to replicate his results, and she turns her shopping lists over to the system. With Bob freed up to handle other tasks, Alice has been thinking that she has been spending so much time thinking about shopping lately that she really needs some help to pick up the slack at her job.
2.3 How To Know When Your Systems are Down and What To Do About It The Second Something Happens

At her work, Alice is responsible for making sure that all of the company’s services stay up and running. One of the problems that Alice has is that she usually does not know if something is broken until someone needs it. Sometimes that can be quite quickly, but other times a service can be down for quite a while before anyone notices.

In either case, Alice usually discovers that something is wrong when someone wants to use the service “right now.” And so fixing whatever is wrong is very stressful because there is someone (or several someones) all waiting for it to be fixed. Alice decides that she would be happier if she could start fixing things as soon as they break, and so she decides to set Bob to work to make her life easier at work.

Alice brings Bob in, and sends him first to meet with everyone who uses the company’s services. Bob makes a list of all the things people do with the services, how often they do them, and how many people do each thing. From there Alice and Bob make a list of activities that replicate all of the things people do with the services, and Bob makes a schedule to test each activity with a higher frequency than people would do in their normal days, doing the most important ones first and most often.

Bob uses the plan to start testing all of the company’s services. Any time a service is not working he alerts Alice and she is able to fix things, possibly even before people notice they were broken.

With Bob’s help Alice’s stress levels go down and she is able to fix things better because she has more time to plan and carefully solve the problem rather than, “getting it working for
right now”. One problem Alice does have though was that her work time is more fragmented since Bob comes to her each time he finds something broken (and since he is proactively looking, he finds things more often then people did before).

Alice looks at the list of things Bob is testing, and at the list of things that he finds that need to be fixed. She realizes that there are several categories of problems that Bob finds: quick fixes, slow fixes, real problems and real problems that can be ignored with a quick fix. Quick fixes are the problems that can be fixed by rebooting a computer, changing a setting or running a command. Slow fixes involve making changes to some aspect of the system that takes special knowledge and usually extra time. Real problems occur when the system does not what it is supposed to do and will have to be redesigned. Fortunately Bob finds very few of these, and they tend to come from the people having new ideas about how to use the systems. What does happen occasionally though are the real problems with quick fixes. In these cases, the system should be fixed, but a quick fix will allow the people to continue to work for now.

In particular, Alice notices that there are some frequently occurring quick fix problems that are almost always noticed by others between the time Bob sees them, and the time she can fix them. Alice discusses it with Bob. Bob is not an expert like Alice, but he is capable of doing several of the quick fixes. Together they prepare a list of quick fixes that Bob is happy to do before he lets Alice know about the problem.

Having Bob quickly solve some problems works quite well. The systems have a much better uptime than they did before, and Alice and the people she works with are much happier and can get more done. As with her two systems at home, Alice starts to think about how she could write a program to replicate Bob’s work.

Alice realizes that this situation is different from the first two systems. Previously, Bob was only responsible for obtaining information, using it to make decisions, and letting her
2.4. WHAT PEOPLE WANT FROM PERSONAL AGENT SYSTEMS

know which decisions he had made. This time, Bob also has to make changes to things in
the real world as well, and her system will need to be more complex.

Alice implements the system by encapsulating the information Bob was getting from the
test runs. Then she writes programs and scripts to use the different tools Bob uses. Finally
she combines it all and has a system that monitors her systems.

The system needs to be tuned to some extent, as Bob often makes small changes to his
schedule based on his knowledge of each day. In order for the system to be as efficient Bob
needs to be able to explain the different situations and how he reacts differently to them.
Over time the system also needs to be maintained with new insights from Alice and Bob to
ensure that it responds correctly to situations that people can use their intuition to problem
solve.

2.4 What People Want from Personal Agent Systems

The problems that Alice is trying to solve could be viewed as superficial. As an experi-
enced IT professional, Alice would probably be aware that monitoring software is commonly
available and system administrators have been creating scripts and tools since the dawn of
electronic computing. Equally, anyone who has lived in the same place for any length of
time knows how to adjust to the weather to arrive at work on time, or what the best way to
get the groceries they want is.

Simple as these problems may seem, they show why personal agent systems can be
important. In each situation, Alice wants to bring together disparate information that is not
designed to naturally fit together and she wants to bring them together in such a way that
she can use them for her own purposes.

One thing to note is that in each of these scenarios there is an example of refinement over
repeated configurations of work for Bob. While it is conceivable to apply machine learning to this step of understanding a problem, it is important to understand that the primary learner in this situation is Alice, understanding how to better construct and organize her assistance system.

One benefit that Alice has is that she is a programmer. She is able to write programs to solve the problems she was able to fix with the help of Bob. A composable personal agent system, like systems using the Task and Workslip Scheme, would allow more people to generate solutions like Alice, regardless of their programming skill.

The following chapter looks at the context for personal agent systems, how they are used currently in the real world and how they fit into the academic context of multi-agent systems. The situations from this chapter will be used to help illustrate and provide context for the definitions and philosophical guidelines that comprise the scheme that will be presented in later chapters.
Chapter 3

The Context of Personal Agent Systems

Artificial intelligence techniques are used in numerous popular applications to help people. Current examples include Apple’s “Siri” [Appb], the “Google Assistant” [?], Microsoft’s “Cortana” [Mica] Amazon’s “Alexa” [Ama] and Mycroft [Mycb]. These all combine improvements in processing power, network communication and AI techniques to provide assistants to people in their own homes (or their own pockets). Beyond the popular applications, personal agent systems have been explored in academia for many years with many projects looking at the mechanisms for people to work with and rely on agents. Further, there has been significant research into the engineering of agents and personal agent systems.

This chapter provides context, for the world in which the Task and Workslip Scheme for Personal Agent Systems fits. It explores some of the history of artificial intelligence and multi-agent systems in general, to provide an understanding of the roots of these systems. It then focuses on the current growth of popular assistants, such as Siri, then on the development and applications of several well established academic frameworks for engi-
neering personal agent systems. Finally this chapter includes a discussion of a few different approaches to constructing personal agent systems.

3.1 Help from Artificial Intelligence

Even from the very early days of research into artificial intelligence it was understood that AI techniques could be used to help people interact more effectively with computers. The work proposed by McCarthy and others for the Dartmouth Summer Research Program [MMRS55], shows that there was already interest into how to interact with computers using natural language. Much of their work was focused on establishing the basics of artificial intelligence. However, even in this establishing event there was already a glimmer that AI techniques could be used to make computers easier to use for people, with several projects relating to cooperation between people and agents.

Beyond the significant research into how to process natural language [SJ01], research also continued into what aspects of language were even necessary. Weizenbaum’s work with ELIZA, an agent capable of communicating with a human, showed that interaction could occur without the agent having any contextual knowledge at all [Wei66]. Participants reacted to ELIZA in such a way that it showed people were willing to engage with an entity that seemed like a person, even if it was really not. On the one hand, this supports the concept that personal agent systems will be accepted by people, but on the other hand, there are, justified, concerns about the systems that people will trust [LH15].

As Artificial Intelligence work continued, systems were developed to help support people making decisions such as XCON which used rule systems to help determine which VAX11 system components were needed to satisfy a customer’s need [McD88]. Similarly, systems were developed to attempt to encapsulate expert knowledge in such a way that it could
3.1. HELP FROM ARTIFICIAL INTELLIGENCE

be used to support people working in the field, particularly medicine with the example of MYCIN [SB75].

These systems proved to have operational and sociological problems. It was difficult to properly encode expertise and further to convince people to share or use that expertise. However it shows that there was interest in finding ways to assist people using Artificial Intelligence techniques.

The interface of Artificial Intelligence has also strongly influenced where these techniques have been useful and accepted. The Microsoft Office Assistant “Clippit” (or Clippy) [Micb] was not well received because it presented an intrusive interface [Mey15] [SPK16] and did not have a very broad selection of useful behaviours. In other circumstances, AI techniques have been widely accepted such as the evolving role of recommender systems in the majority of e-commerce and media websites [RRSK11]. This is often because the people using the system are able to accept or reject the recommendations made by the system and then, ideally, have the system tune itself for better future recommendations and because the system hides its “intelligence” behaving much more like traditional software rather than an active assistant.

While this suggests that people might inherently reject having an AI assistant, it better follows that these assistants have to be constructed in such a way that the people using them remain in control and in such a way that their interface is not intrusive. The rise of Artificial Intelligence Virtual Assistants, such as Apple’s Siri show that people are interested in having an assistant that helps them with routine tasks, so long as they retain control and benefit from the system.
3.2 How Siri and Friends Moved In - Personal Assistants in everyday lives

As far back as 1987, Apple Inc. [Appa] has been interested in demonstrating how Artificial Intelligence should be applied to peoples lives. Knowledge Navigator[Appc] was a demonstration of how Apple viewed the potential of computers in higher education [Col11]. Their intention was to show that the benefits of a computer system and telecommunications technology could be encapsulated in such a way that they seamlessly fit into a person’s life in the same way a personal assistant should. In their demonstration they showed the system notifying a researcher of a new paper he would be interested in, leaving and taking messages and phone calls, reasoning and extrapolating from data and instantly searching for information based on informal cues.

As it is obvious from the original demo, in 1987 most of the techniques and hardware needed to actually implement an Artificial Intelligence personal assistant were not ready. Over the next 30 years AI techniques such as Speech-To-Text (as well as Text-To-Speech), and Natural Language Processing had to develop to allow an agent to interact with people.

While the exact construction of many Virtual Assistants is kept as a proprietary knowlege, the open source assistant Mycroft exposes the modules necessary to create this type of software [Myca]. In addition to the above there is a need for Intent Parsers to link the spoken phrases to the Skills which the system can carry out. There is also middleware needed to allow the different components of the system to work together. Skills allow systems to do basic tasks, work with systems like e-mail, calendars or the Internet of Things. However for really “intelligent” behaviour, such as being able to answer how many days old a person is, requires that these systems communicate with another Artificial Intelligence service such as IBM’s Watson[Int] or Wolfram Alpha[Wol] which are able to collect and link data and
3.2. *HOW SIRI AND FRIENDS MOVED IN*

concepts to produce structured “knowledge” through linking words, concepts and facts. Linked together with a personal agent these systems produce an assistant which can roughly behave in the same way a human assistant might.

The growth of these systems shows that people have a continued interest in supplementing their use of their computer systems using AI techniques. At this point in time, this is primarily offered as an easier interface to existing services which the technology can already perform. As development continues, if the trend continues as it has, then these systems will be able to do increasingly complex and useful things.

While this will have obvious impacts improving the quality of life for people using these systems, there are also potential risks in terms of people’s privacy and autonomy. It is possible for a personal agent to leak information about its principal in an intended fashion, such as by not verifying that it is giving information to its own principal[Tre15].

Knowledge-linking systems, such as Watson or Alpha require information to link and need more information to work better. Consider asking an assistant how many days old you are. This requires that the system knows both your birthday and how to count the number of days between then and the current date. If it is missing either of these capabilities then it cannot do the task. Therefore with more data there are more potential tasks that a personal assistant can perform. However this comes at the cost of further exposing personal information to whichever services hold that knowledge. To this date, there do not seem to be cases of privacy leaks directly from these services, but the risk of leaks remains.

Finally, systems such as these often leave the principal at the hands of an outside organization when decisions are made regarding which services and skills the system will have. Consider a principal asking “What are the scores?” This requires the system to make inferences about the principal’s intentions: which sport? which league? which teams? which time frame? and so on. It can be costly to develop the skills for these systems, and it may
not be cost effective for an organization to create a skill if the uptake is low. However this reduces the usability of the system for the principal.

It is possible that outside organizations might make agreements which alters the agent inferences about the principal’s intentions. “What are the scores?” might be open to the highest bidder even if that bidder’s scores are not the ones the principal is interested in. Open Source systems such as Mycroft improve the transparency of situations such as that last case, but the fact remains that unless a principal can control all of the aspects of their personal agent system, there is always a chance for a reduction of their autonomy.

### 3.3 Personal Agent Systems and Personal Agent System Frameworks

Personal agent systems have been a topic of research since at least the mid 1990s. Early systems, such as COLLAGEN[RS98], sought to find a system for principals to work with agents to accomplish specific tasks. Since early days there has been a struggle to combine the agent as task-assistant with the agent as task-executor. There has been a lot of work needed to manage the privacy and autonomy needs of principals with the effectiveness and efficiency of agent systems using AI techniques. This section contains a description of several key personal agent systems and approaches to constructing personal agent systems, to provide an understanding of the design and application of the work.

**Electric Elves**

The Electric Elves was a project at the University of Southern California that began in 2000 and continued for several years [PTA+00] [CGK+01] [CGK+02]. The project was intended to
combine different Artificial Intelligence techniques into a larger and more complex system, and to look at the impact involving personal agents would have on people and organizations. In particular they addressed five challenges in creating agents to work with human organizations: adjustable autonomy of agents, reliable access to information, matchmaking of interests and capabilities, coordination of agents and robustness and scalability of agent systems.

The Elves supplemented researchers working together in research groups. Each researcher in the group had a proxy agent assigned to act for them, their “Friday”. Additionally, there were agents that represented outside groups and services and agents to coordinate agent activities. The systems were designed to handle organizing a weekly research meeting for a research group, and ad-hoc meetings inside and outside of the research group.

Each Friday has access to their principal’s\(^1\) calendar, knowledge about their location and a method to communicate with them (using a hand-held GPS unit and a Palm Pilot respectively). This allowed the agent to assist their principal by identifying situations such as where the principal was not in the correct location for a meeting. The Friday was able to act by first querying the principal as to their intention (delay the meeting a few minutes or cancel it) and then either relaying that response or acting in their principal’s stead if the principal was not able to respond in a timely fashion. The agents used different learning techniques to help ascertain what their principal would want to do.

Fridays were also responsible for coordinating the regular group meeting, including coordinating the time it was being held, selecting a speaker and arranging for food to be delivered. As with ad-hoc meetings agents were able to select time based on their principal’s calendars. Agents then conducted an auction to determine who would speak, based on the

---

\(^1\)The term principal is applied to keep the terminology consistent with the remainder of this document. The authors used the term user, but principal continues to better define the relationship between the agent(s) and the person(people) they are working for.
principal’s willingness, capability and the time since the principal had last been the speaker. Once the meeting was scheduled, the Fridays would then forward food preferences to an external proxy agent which would then fax orders to local restaurants for delivery.

The Elves were also responsible for monitoring travel arrangements when visitors were travelling to meet the group. The agents were able to monitor visitor’s flights and update the group, and manage the visitor’s meetings, depending on delays in travel. If the visitor was arriving early the agents would also send a fax to the visitor at their hotel with a welcome and information about visiting the area.

Finally, the system was also capable of organizing meetings outside of the research group, particularly ad-hoc meetings at conferences or workshops. Agents were responsible for combing the research interest information available regarding other participants and matching that information with the interests for their principals. This included checking an internal database of information about researchers, but also outside sources such as external databases and webpages. Once compatible interests were found the agents would send messages to potential participants and then find a suitable restaurant and make a reservation there.

Over several iterations of the Electric Elves the authors have addressed the challenges they faced in several different ways. Adjustable Autonomy of agents required that the Fridays be able to act on behalf of their principals when the principals could not act. To determine how to act, agents applied machine learning techniques. Their original application of decision trees produced overly strong generalizations that caused agents to commit to decisions that their principals did not want. Following applications used a significantly more adjustable system which uses Markov Decision Processes \cite{Put94} and which continuously reevaluates the action that it should be taking at any point in time. Actions were also prioritized based on their impact, so it was preferable to defer taking an action, so long as it had a limited
3.3. PAS AND PAS FRAMEWORKS

impact, then once there was an impact choosing the most “manageable” alternative for the context. For example, if the principal seems near and is sometimes late to a meeting like this one, defer the meeting by a few minutes or if the principal is not near then a cancellation would cause the least amount of work for all involved.

Additionally agents had to be able to ask other agents to act as proxies\(^2\). For example, when ordering food, the Fridays deferred the actual work of sending messages to local restaurants to specialist agents designed to do this type of work. In order to manage situations where an agent was not able to handle a task itself, agents had to be able to find another agent which could do this task or find a group of agents which could work together to handle a given task. Agents used a matchmaking system [GR01] which allowed them to define their capabilities and query about those capabilities through a formal language. This language allows the decomposition and unification of tasks to allow the match maker to rewrite the query so that it can find an agent or set of agents which fulfill the query.

To handle the challenge of reliable access to information, the agents use machine learning to generate wrappers that extract unstructured information from various sources and render it in a common form that the agents can use. For example, the system interprets research interests for a person based on their publication record. Various wrappers are generated which take information from publication databases and combine it into a common format for a matchmaker. The matchmaker is then able to use this information when asked to find people to meet with when asked to set up an ad-hoc meeting for a researcher.

Agents in the Electric Elves have diverse domain specific responsibilities, such as principal tracking, external communications, wrapping external information or matchmaking. To allow agents to work and reason together to get work done, the agents need a scheme to allow

\(^2\)This use of proxies also allows the Electric Elves system to be heterogeneous in terms of agents and agent technology by having communication proxies for those agents which need help communicating.
them to work together. The agents in the Electric Elves use a teamwork based integration architecture (TEAMCORE [PTCC99]) which allowed agents to identify, commit to and complete goals, handle failures, allow agents to communicate as necessary, regardless of design, and generate tasks for agents to complete. This provided a common execution component with communication abilities and reasoning, which could be wrapped around any type of agent regardless of its design and allows agents to communicate through a common shared medium.

The robustness and scalability of personal agent systems was ensured through the teamwork architecture and the ability to use matchmaking agents to manage the execution of tasks that need to be completed. Agents were able to fail in their tasks and the system was able to reformulate the task the agent was completing and issue it to other agents (if possible). While the open communication structure allows for some scalability, the system generated a large amount of messages (on the order of tens of thousands) which might present a problem in a larger group of people.

When the creators of the Electric Elves deployed the system in their own research groups, (over several periods of several months, beginning in June 2001) there were some problems that arose. The authors discussed these in a later retrospective [TBP+08]. Some of these related to technical problems, such as the difficulty (especially in early iterations) that Fridays found in learning intentions from their principals (for example an agent learned to cancel meetings by moving them back in five minute increments indefinitely).

Other problems which affected the Elves were more social in nature. In particular there were issues related to privacy and social norms. The Friday agents made it simple for principals to see the location of another principal. For example if a principal was en route to a meeting, but stopped to buy a coffee while saying they were “stuck in traffic”, other principals were able to see exactly where the principal was. Similarly, it was not possible to
close the door of your office and leave the lights off to get uninterrupted working time.

There were also issues with how the rules the principals and system developed were applied. For example there were issues where students felt they were being treated as “lower priority” because their meetings with professors were being cancelled in favour of the professor meeting with someone else (although the professor claims these were primarily issues of scheduling). Principals would also alter their calendars, adding more entries, because the meeting speaker schedule was affected by how busy the principals were, so adding “Play Basketball” to your schedule reduced your odds of having to speak.

The social issues that arose around the Electric Elves are due partly to the experimental nature of the system. People agreed to participate without a clear understanding of how the system would impact them. It is easy to see the benefit of having a system to automate scheduling the weekly group meeting, but harder to predict the privacy implications that might arise.

These problems highlight that before anyone uses a personal agent system there needs to be a clear and frank explanation about the implications using a system will have. Developers of these systems need to ensure that the system is able to adapt to and reflect the intentions of the people using the systems.

The Electric Elves provide a prime example of early systems involving personal agents. They illustrate the necessity for a common architecture for communication and coordination of agents. They also show how primary problems, such as those of how a principal and an agent can act together, need to be addressed. Additionally, they provided a test bed for several technologies useful for helping agents to act autonomously to support people.

Some of the aspects that caused problems with the Electric Elves was the intention to apply them to the task of group coordination. Focusing a personal agent system on a single principal reduces the societal problems because there is no internal risk of privacy leak. It
is easier to ensure that each principal has a full understanding and willingness to use the system and to ensure that the agents are acting for, not on the principal.

**CALO**

While the Electric Elves focused on incorporating Personal Agents as a communication and organization tool for a group of principals, another system called Cognitive Agent that Learns and Organizes (CALO) [SRI] focused on ways to assist a single principal with common tasks and workflows\(^3\). As with the Electric Elves, the CALO project looks to find ways to take advantage of Artificial Intelligence techniques, both as a way to improve the work the agent does for the principal, but also as a testing ground to improve the techniques.

When using CALO, the agent resides within the work environment of the principal (particularly, as an element of their operating system interface and the programs they use for their usual work as a “knowledge worker”). The agent acts as an assistant to the principal, providing reminders, gathering information, scheduling meetings and completing specific tasks that can be accomplished by (components of) the agent.

As an example, the agent working for a Principal Investigator might receive an e-mail from a research manager with a request to purchase a new laptop for a summer student. The agent could then fill in the requisition paperwork, and upon realizing there were insufficient funds, suggest the principal have a meeting (with several proposed scheduling alternatives) with the research office which is providing grants for research equipment. In this way, the agent is expected to reduce the principal’s work by handling routine and defined tasks and consulting the principal as necessary for the completion of these tasks.

Early iterations of CALO [YSSMM09] focused on a delegative model of assistance, in

---

\(^3\)The CALO team use a variety of different terms for the agent(s), but converge towards personal agent or personal assistant in later papers and refer to the principal as the user.
which the principal provided the guidance to the agent and the agent only executed tasks on the instruction of the principal. Later iterations of CALO [YSSMM12] focused on a proactive model of assistance which (more like the example above) allowed the agent to act in such a way as to take “safe” actions in support of the principal.

Structurally, the CALO agent was built out of several modules, or sub-agents which worked together to support the principal in the execution of several different tasks (and were each built out of further modules or sub-agents). These modules included the process controller (SPARK [MM04]), the task manager (the Personal Execution Agent, PExA) [MYS05] [MBB+07], the scheduling manager (PTIME [BPC+06]) and the information manager (IRIS [CPG05]).

The information manager was responsible for providing a common knowledge base and ontology (CLib [Kno]) for the other modules to use. It also provided an interface for the principal via various plugins in the principal’s productivity software and some other specialized programs that used the system directly. This interface allowed CALO to collect information for (and from) the principal to help the other modules function with more information.

The scheduling manager provided an intelligent system to assist the principal in scheduling meetings (and other events). The manager was able to reason over the principals schedule and provide suggestions on potential times for events that would best suit the principal’s existing schedule, predicted workload and preferences. The manager would offer several alternative schedules to the principal, allowing the principal to determine which was best or to override and enter a preferred schedule. Over time the manager learned the principal’s scheduling preferences and was able to suggest better alternatives.

The task manager was able to work through the information manager to execute tasks for the principal. In its basic state, it worked as a “to-do list manager” for the principal, keeping track of the things the principal needed to do. As it was augmented with more
knowledge it was able to create models about tasks, such as which dependencies needed to be met before another task could be worked on. The manager also used “workflow” models to track which steps needed to be done for a particular task and as it was augmented with further abilities (through a plug-in system) it was able to do certain steps (such as fill in paperwork) for the principal.

The process controller coordinated the actions of the other modules. This coordination came in the form of an augmented Belief-Desire-Intention [RG91] framework. Tasks were reframed as potential goals for the agent. These potential goals were then weighed to determine which actions the other modules could take would produce the most overall benefit for the system. These potential goals would then be transformed into the intentions / actions of the system. The goals could be provided by the principal or generated out of other actions the system was taking. As the system transitioned from the delegative model to the proactive model the amount of goals the system was able to generate for itself increased.

The creators of CALO wanted to create a system which would be directable, personalizable, teachable and transparent [MBB+07]. It needed to be apparent to the principal that the agent worked for them (and not the other way around) and that the agent would learn a model of the principal’s preferences and adapt its behaviour to that model. In order for the agent to learn it must be simple for the principal to communicate with it and equally it must be simple for the principal to understand how (and why) the agent made the decisions that it made.

These goals lead to the initial delegative model of interaction, which ensured that the principal was in control, rather than force the principal to conform to a model provided by the agent. While this seems to have reduced discomfort with the action of the agent, early iterations of the system had problems in finding ways for the agent to provide assistance beyond what the principal could do themselves and, particularly, in interacting with the
principal in a convenient way.

The creators of CALO tested many different interaction modalities before settling primarily on an “Instant-Messager” style interface which allowed the agent to interact with the principal via text query, augmented with decision interfaces. The principal was also able to query the agent with regard to its knowledge and decision making process. This was supported with other interfaces which were appropriate to the task at hand (for example if the agent was filling out a PDF form for the principal, acting as a plug-in within the PDF reader). Together these produced an interface that allowed the principal a high degree of control over the agent’s activity.

In later work [YSSMM12] the creators of CALO sought to increase the proactivity of the agent, allowing it to have greater autonomy, but requiring that it still work in a way that satisfied the principal. To achieve this they implemented a “Theory of Proactivity” which included the sub-theories: a “Theory of User Desires”, a “Theory of Helpfulness” and a “Theory of Safe Actions”. These provide, respectively, a model that the agent could use to determine what the principal wanted to do, a model that the agent could use to determine what actions it could take to help the principal do what they want to do, and a model that the agent can use to determine which actions it is capable of doing in a way that could help the principal.

These models along with a framework of “Assistance Patterns” allow the agent to take actions to help the principal. These components are intended to allow the CALO agent to demonstrate the creators’ principles for proactivity, namely that the agent should be:

- valuable (advance the principal’s tasks, in the principal’s opinion),
- pertinent (relevant to the current situation),
- competent (able to do what the principal needs),
3.3. PAS AND PAS FRAMEWORKS

- unobtrusive (able to work without interrupting the principal unnecessarily),
- transparent (understandable to the principal),
- controllable (working the way the principal would like),
- deferent (“gracefully unimposing”),
- anticipatory (aware of the principal’s current and future needs),
- safe (minimizing negative consequences).

These principles are necessary to ensure that the agent remains useful to the principal when acting proactivity. This helps the agent avoid many of the problems that the principals experienced when working with the Electric Elves. Combined with the restricted focus of one agent acting for the benefit of one principal, this clarifies the role of the agent as an assistant to the principal, and helps the principal trust and rely on the agent.

It is difficult to ascertain the overall technical and social success of CALO from the papers written about the many components of CALO over the project period. Generally, improved techniques in machine learning have helped the agent to adapt better to the principal. Improved AI techniques have helped the agent to act more “helpfully” in general, and modern software and hardware have made it simpler for the agent to integrate into the principal’s work environment.

RADAR

The Reflective Agents with Distributed Adaptive Reasoning (RADAR) project focused on finding ways to usefully improve day-to-day use of the computer as a management assistant
The intention was to add intelligent assistance to a principal’s\textsuperscript{4} existing workflow and tools. The system used specialized agents, which were able to assist with specific tasks, to make the principal’s work easier. Rather than create new interfaces for the system, RADAR uses bridging agents to connect its internal agents with existing software.

The work with RADAR was largely focused on developing tools to assist with task management. Generally, the role of the system was to identify tasks that the principal needed to do, and help organize frameworks for doing them, particularly helping to “triage” tasks into high and low priority, and batching similar tasks so the principal would have to switch contexts less frequently while working. Through learning, the system was able to assist on tasks which could be automated, or partially-automated, to assist the principal.

In this case the creators of RADAR defined a task as:

a unit of work that the user cares about that can be automated (or partially automated) by RADAR. The unit of work could be assigned to a single task specialist, or it may involve the coordination (through a task planner) of multiple task specialists. Such a planner would itself be implemented as a specialist.

In this way, a task was any action that the principal was able to do (with the assistance of RADAR) and tasks would arrive over the course of the principal’s day either as a result of outside action or as a result of the principal’s own work. Examples of tasks agents were able to do include: responding to e-mails, filling in forms, updating a website or scheduling meetings.

To assist the principal and integrate with their workflow RADAR included several different types of agents. The primary two types of agents were the Task Manager Agent and\textsuperscript{4} Again, the creators of RADAR refer to the user, rather than the principal, and the system as the Personal Cognitive Assistant
the Task Specialist Agents. The task manager agent was responsible for determining what
tasks needed to be done and assigning those tasks to the task specialist agents. The task
specialist agents had an action component and a learning component. First they acted on a
task for the principal (in the way the principal wanted the task acted on) and second they
learned from the principal better ways to act on the task.

Further, there were bridge agents that would connect other software to RADAR and
transform information. Agents would communicate via a knowledge base. Using bridge
agents extractors and abstractors, which were able to take information from outside sources,
and structure it into useful knowledge for the agents.

One example of how RADAR might work is the task manager noticing a new task arising
from the principal’s email. Fred would like to schedule a meeting with the principal, John,
and Melinda. The extractors would process the email and determine the basic information,
such as, that the task was for a meeting, and the people involved would be Fred, John,
and Melinda. The Abstractors would then be able to extend this information, such as that
Melinda was John’s manager and that John preferred to schedule meetings with Fred in the
morning. Once the data was added to the knowledge base, the Task Manager would become
aware of the task to create a meeting and would assign the work to the Meeting Scheduler
Specialist which would use its own approach to try to schedule the meeting and then possibly
offer options to John to decide which meeting would work best.

Using this technique RADAR agents can help the principal extract meta-information
about tasks and which tasks need to be done without the principal having to do the “first
pass” work of finding the task and prioritizing it.

Evaluation of personal agent systems is often very difficult. In recognition of this the cre-
ators of RADAR designed an experiment in using RADAR technology, that could be applied
longitudinally over the course of development of the system (more than 5 years) [SBC+06]
In this study novice users of the RADAR system would be responsible for responding to an “artificial crisis,” where the study participants must help an academic conference move from one venue to another without the assistance of the original organizer (although they had access to his data, in the form of a RADAR knowledge-base). Participants in the study are responsible for handling tasks submitted by e-mail and must identify not only which tasks need to be done but what the most efficient way to do the tasks is. Over the course of the project over 700 people participate in either “main” annual studies or interim studies.

The results of this study showed that the principals were able to solve tasks more efficiently and effectively with the help of the system. It also showed that integrating the intelligence of the system into the user interface and providing very limited insight into the decision making process of the agents did not hinder the principals working with them.

As with the approach used in CALO, the approach used in RADAR has several improvements over the results seen with the Electric Elves. In particular the reduction of scope from a group of principals to a single principal greatly improves the privacy and functionality of the system from the perspective of one of those principals. RADAR does have the concept of a shared knowledge base, but includes a Security Manager that helps to restrict how information is shared.

Functionally CALO and RADAR are quite similar in their intent, to “augment the desktop” and make it easier for the principal to do their job. Both search out tasks from the principal’s environment (which is effectively their e-mail) and has systems for one or more agents to act on those tasks and do something “automatically” for the principal. Both provide ways for the agents to integrate into the principal’s tools and both provide a further interface for the agents to query and interact with the principal. RADAR seems to have relied on, and needed, this less than CALO.
Interestingly even though RADAR has a structure more matching one-agent-to-one-task, the same developers would write the bridge agents necessary to extract and abstract tasks as well as the agent which would act on the task. The authors of [GS07] note that the developers resisted the necessary encapsulation of detection from action feeling that it would be “more efficient” to do both parts in the same piece of software. This is interesting because it shows a reaction that will have to be overcome when designing any multi-agent system to support people using a more modular and encapsulated system.

**ANTICO**

The Anytime Cognition Personal Assistant concept, ANTICO [OMSK11] [OMS14] (originally ANTicipatory Information and Planning Agent ANTIPA [OMS10]) is a project intended to create agents similar to the Friday agents of the Electric Elves. In particular, the creators focused on the requirement that Friday agents be able to provide necessary contextual information to the principal\(^5\), hopefully improving the principal’s ability to create plans in stressful situations. Rather than developing directions for the principal to follow (such as the RADAR approach), ANTICO is intended to operate as a support for the principal, adjusting its support as it observes the actions of the principal.

In a rough abstraction, using ANTICO a principal has a single agent, which is responsible for supporting them while they do a complicated or stressful job. Each ANTICO agent has four components, an action observer, a planner, an information gatherer and a information presenter. ANTICO is intended for use in high stress, but well defined, situations. Two situations they use as examples are, peace-keeping and emergency crisis response [MOC+11] [MKC+12]. In these situations a great deal of pre-planning has already been done (negotiations between entities, emergency response planning, etc.) and as such it is easier to define.

---

\(^5\)Referred to in the ANTICO literature as the user, or as the planner
what situations should occur in which order and which actions a principal is likely to take to lead to those situations. Agents model these situations and rules as norms and situations within its internal definition language, ANTICO Domain Description Language (ADDL).

Agents in ANTICO observe the actions of their principal and use those observations to inform a model of the principal’s intention in the form of a Markov Decision Process. The agent then uses the model to construct a plan of what the principal is most likely to do and what the agent can do to support the principal’s future actions.

The plan is constructed as a tree, including in each node the task the agent can do (retrieve information, communicate with another person or agent) the probability that the task will be useful to the principal according to the current model, and the deadline by which it must be done to be useful. As tasks are finished, and the principal needs the results, the currently relevant and available information is presented to the principal. This limits the amount of information presented at one time to limit the principal’s information overload in an intense context.

As with most other personal agent approaches to follow the Electric Elves, ANTICO addresses one of the primary sources of problems, primarily that using the agents in a group context created unwanted social pressures and interactions. With the Electric Elves, the principal was forced to follow the direction of the agent because the agent was unable to intuitively respond to the principal’s intentions.

ANTICO leverages well described situations to allow system developers to create agents using ADDL descriptions of the problem domain and the actions the principal is likely to take (and possibly should be highly encouraged to take). This allows a fair amount of adaptability on the part of the agent, although much of their evaluation presents the system in such limited situations that the agent should really have no problem determining what tasks the principal will take and which information they will need.
### 3.3. PAS AND PAS FRAMEWORKS

**Tropos, Xipho and Arnor**

While not frameworks for constructing personal agent systems, like the above systems, Tropos [BPG+04], Xipho [MS14], and Arnor [AMGS17] are agent software engineering methods. They are designed to provide an approach to help developers design and implement personal agent systems. Of these three methods, Tropos is the broadest, intended as a general approach to designing multi-agent systems. Xipho extends Tropos to apply specifically to the design and implementation of “Context Aware Personal Agents”. Arnor extends Xipho to enhance an agent’s awareness of social context, and principal’s privacy expectations.

Tropos extends traditional approaches to software engineering to assist developers in designing and implementing software using multi-agent system practices. To achieve this, they step back from the standard models of requirements gathering, instead asking how the completed system should behave for all people involved with it. They use BDI modelling to describe how agents within a system will operate, and then define “mentalistic notions” which define how people will expect to interact with the system.

A Tropos model includes actors (people and agents), hard and soft goals, plans and capabilities. The model includes goals for various of the actors, plans that can be enacted to satisfy those goals and capabilities that will be needed to enact the plans. For example a principal may want to have a ringer agent manage whether or not their phone will ring. In general, they do not want to be disturbed while working, but do want the agent to disturb them if the phone call is an emergency. In this case the principal and the ringer manager are the actors, the soft goal is to not be disturbed, and the hard goal is to be notified of an emergency. The plans are to divert a call or ring the principal’s phone and the capabilities...

---

6 The authors use the term stakeholder to refer to any person interacting with the system. They further define a primary stakeholder to be the person for whom the system works, referred to here as the principal, and secondary stakeholders as people affected by the system.
are the agent’s ability to do either of those two activities.

One interesting aspect included in the example models in [BPG+04] is the actor perspective, which illustrates that different actors in a system will have different understandings of the system in terms of what goals should be met, which plans should meet them and which capabilities are available to enable them. They provide an example of a museum information system, and show how the museum visitor and museum curator, have very different sets of expectations for the behaviour of the system.

To design a Tropos model, the authors use an iterative approach, determining first what plans are needed to meet the goals within the system, then which actors can best enact those plans, and finally how the capabilities those actors provide should be connected to allow the plans to be enacted.

A drawback of the Tropos model is that it is difficult, or at least complex to define the conditions under which certain plans should be enacted. Xiphos extends Tropos to help mitigate this problem by introducing context to Tropos models. This allows the developer to identify different situations in which different plans should be enacted to meet the goals of the system. As an example, this allows the Ringer Manager to determine, rather than just which soft goal should be violated to enable a hard goal, what situation the principal is in and therefore which goals and plans should be the focus of the agent.

Xipho models allow further decomposition of Tropos models. In particular it allows a developer to decompose plans that have alternative actions. It allows the developer to manage conflicting goals in the model, allowing them to depend on context. It also allows the developer finer control over the ‘softness’ of goals, so a goal to not be disturbed is more easily broken with family than it is with colleagues, or vice versa.

Capturing context is a difficult task. Xipho calls for utilizing agent’s Beliefs to define the elements of context which are important. So for the Ringer Manager, the developer must
capture and the agent must identify where the principal is and what the principal is doing, and potentially how the principal is feeling about what they are doing.

Arnor extends Xipho by using a socially aware approach to guide the design of agents within a system. In particular Arnor uses norms [CJA09] to create a set of rules that define the acceptable behaviour of agents within the system. These norms allow the developer to describe at a finer granularity how agents should act within particular contexts. Taking advantage of these norms allows a developer to create a system that better protects the privacy not only of the primary stakeholder (the principal), but also those around them. In operation, norms provide a higher level set of world information above an agent’s beliefs. Norms are modelled as tuples of an actor, an object, an antecedent (describing a situation), and a consequent (describing what will happen if the antecedent is true). This allows a norm to describe a behaviour the system should have with an associated reward, or a behaviour the system should not have with an associated penalty.

To integrate this social expectation into their agents developers must first follow the steps needed to generate a Xipho model. First they must model the actors, goals and plans within a system and then extract contexts to identify alternate plans and conflicting goals. With the contexts established, the developer must identify the norms at play in each context, and potential conflicts between those norms. Where conflicts exist, the developer must refine the contextual information to determine which norms will be followed. Finally the developer must determine what the principal will “experience” through the agent, identifying which actions with enhance or detract from the principal’s experience of interacting with the agent.

These three software engineering models for agent systems provide a way for developers to focus their efforts while designing agent based systems, and particularly personal agent systems. The authors’ experimentation shows that these techniques make developing systems more standardized (in terms of time and effort), if not strictly improved.
3.4 Methods for Agent Engineering and Communication

While there are many different paradigms that exist for constructing agents in multi agent systems, this section provides a discussion of three particular methods that are important to consider either as a direct influence or a counter example for the Task and Workslip Scheme. BDI-Architecture is widely used, including the recent systems discussed previously. FIPA is a widely used communication protocol between agents. Finally, Hive is a unique agent construction approach, which allows smaller agents to exist and share resources in a networked computational environment.

3.4.1 BDI-Architecture in general

One of the most common approaches to constructing agents in personal agent system applications, BDI-Architecture [RG91] provides a general description of how the internal state of an agent can be modelled. This model includes the agent’s understanding of its environment, its understanding of what it needs to do and its understanding of the work which it does. Respectively, these represent the beliefs, desires (or goals) and intentions of the agent.

Each of the beliefs, desires and intentions of the agent are modelled as potential worlds using temporal logic, special modal logics, which illustrates the different actions the agent might take (or might have taken) and how those actions should change the world. These logics are also used to describe the changing conditions of the world that the agent is not capable of controlling.

The agent’s beliefs include what the agent understands to be true, as well as the things which the agent believes will inevitably become true (at least in some of the worlds). The
agent’s desires are conditions of the world that the agent is trying to make true. The agent must believe that it is possible to meet these desires. Generally there is no requirement that the desires of an agent be compatible, however the goals of the agent must be both achievable and compatible with each other.

The intentions of the agent are a subset of the desires which the agent is currently attempting to achieve at any point in time. As time passes the agent is able to update its intentions if it no longer believes that it can (or should depending on implementation) achieve the given desire. The escalation of intentions to the same level of importance as the beliefs and desires (from an implementation detail) allows the agent to explicitly have conflicting desires, but use intentions to produce a consistent set of goals that the agent is currently working to cause to be true.

BDI-Architecture can be viewed simply as a model for agents in multi agent systems, leaving the implementation details independent of the actual logic required to describe the worlds. However, for many designers of multi agent systems (and personal agent systems) it has proved to be useful to use BDI-Architecture as a guide to the actual construction of agents in a system [RG95].

The basic structure of an agent following BDI-Architecture has three data structures to hold its beliefs, desires and intentions. It also has a queue of events that it is receiving from the outside world. The primary algorithm for the agent is to generate a set of potential plans based on its intentions and then select from those plans the best subset of plans that it can accomplish at the current time. It then updates its beliefs based on the events it receives, prunes out any desires that are now either achieved or have become impossible and then updates its intentions as necessary.

The plans that an agent creates include a collection of “primitive” actions which the agent can take, as well as sub-plans which represent collections of those actions. Each plan has
preconditions which must be met in the beliefs and intentions of the agent in order for it to be executed and an invocation, which is an event indicating that the plan should be enacted. In execution of the plans, the agent creates stacks of intentions and corresponding plans. It uses these stacks to allow backtracking along the different states of the world depending on which branches have proved to be true (similarly to how PROLOG uses backtracking to unify its goals).

The selection of an agent’s plans, and the updating of its intentions can be tuned depending on the application of the system and the constraints of the particular instance of the problem being solved. An agent can exhibit realism, strong realism or weak realism, each of which describe a different requirement of the relationship between the agent’s beliefs and its desires. Realism requires that the agent desire all of its beliefs. Strong realism requires that an agent believes that all of its desires are inevitable (there is a path from its current beliefs to the world described in the desire). Weak realism requires only that the agent does not believe that it is inevitable that the desire cannot be achieved. Each of these allow the agent a different degree of flexibility in pruning its beliefs.

BDI agents should also demonstrate commitment. This means that an agent should not update its plans and intentions simply because its beliefs have changed. Instead the agent should only make these updates when it is absolutely necessary to react because its intentions are no longer a valid set of intentions. This allows the agent to be viewed by the principal as more constant in its behaviour, and this also allows the agent to reduce its computational load re-evaluating all intentions for each new event.

The authors of [RG95] point out that one of the primary benefits to using BDI to implement agents is that it allows agent developers to focus on writing the agent plans which can be done in a very high level language. This high level language also makes it easier to modify the behaviour of the system and reduces the amount of time needed for an iteration
of development.

While this provides a good framework for an agent to be built on, there is certainly room for expansion. One particularly useful is the inclusion of soft realtime constraints [VAL10], which expands the intentions and plans of the system to include priorities and deadlines. When selecting intentions the agent now needs to select the set of intentions which do not prevent any higher priority intention from being completed before its deadline is passed. Each primitive action the agent can take is now given a maximum time in which it can be completed. If an action is not completed within its maximum time, then the agent considers the plan failed and reconsults all of its intentions. This further reduces the computational load of the agent and also allows the agent to be much more reliable in a real-world context.

Another possible extension is to use norms as discussed in [CJA09], which provides a higher level of authority for the agent to follow when it is deciding which desires it will select as intentions. Using this higher authority produces a common set of expectations on agent behaviour. This in turn allows system designers to structure the general interactions of the agents (and principals) before handling the specific behaviours and actions of the agents specifically.
BDI-Architecture provides developers of personal agent systems a strong starting point when creating their systems. This is evident in its wide popularity. However, when used as a basis for implementation, it requires that developers manage converting between a very high level logic and efficient operations for an agent. Depending on the context, the flexibility of allowing principals to write plans for the agents in a high-level logic may prove very useful, but depends on the willingness and expertise of the principals to engage. As a system for implementation, BDI-Architecture may prove useful for very diverse agent tasks, particularly in situations where a strong definition of situations and plans already exists, but it lacks specialization that can be achieved with purpose built agents.

3.4.2 FIPA-ACL / FIPA-AIPS

The Foundation for Intelligent Physical Agents (FIPA) [FIP] provides several standards for modelling and implementing heterogeneous multi-agent systems [Pos07]. Of these the standard considered most useful, and most widely implemented, is the FIPA Agent Communication Language (FIPA-ACL) and its expanded version the FIPA Agent Interface Protocol Suite (FIPA-AIPS). These two standards provide a scheme for agents to communicate in a stateful, or contextual, manner.

This communication allows agents to communicate using a number of specific message types (communicative acts) which then use a shared ontology to provide deeper meaning to agent discussions. These include message types such as making proposals, making queries, providing information and negotiating. For example, when booking a hotel room for a principal, the principal’s agent might send a message to the hotel’s agent using the HOTEL shared ontology, requesting a room booking for a period of dates. The hotel’s agent could then respond with an information on price. The principal’s agent could then agree, and
the hotel’s agent could then confirm. This allows the types of messages to conform to the “communicative acts” of FIPA, the message contents to conform to the specific HOTEL ontology, and the actual behaviours of the agents to conform to the wishes of their principals (the guest and the hotel). The messages become part of a conversation, whose data is held by both agents participating, and as such they only need to communicate about the aspect of the discussion they are currently considering.

The intention behind FIPA is to leave the model of any particular agent to the implementer and the principals using the agents. Generally, FIPA is intended to act as an extension of the BDI-Architecture, with agent’s BDI providing the basis for the messages being sent and agents adopting the meaning of messages into their own BDI, but this is not a requirement of FIPA. Because FIPA provides standards for several aspects of agent design (including agent architecture and management) several libraries have been developed to aid developers in creating new systems, for example Jade[Tel] and Active Components(JadeX) [Act] in Java, and SPADE [GPB] in Python.

Similarly to BDI-Architecture for agent behaviour, FIPA-AIPS provides a common way for developers to communicate. In particular, it functions well in cases where a specific ontology has been agreed upon for a domain and has been designed in such as way that it can be used by relatively general purpose agents working in an uncertain environment.

3.4.3 Hive

The Hive system [MGR+99] is a different approach to constructing and arranging agents. Instead of considering the agent to be a unified monolith of logic, execution and resources, Hive breaks that system up into several smaller components and divides responsibilities for behaviours into smaller more flexible agents. Hive is envisioned in an highly networked
environment with a number of different computational nodes where software can be run that can house and execute agents (a container for agents). The agent-housing software is called the cell, and a hive is a network of cells. Within each cell are shadows, resources (such as a monitor, or a sensor) that the cell can make available, and agents.

In the Hive paradigm, each agent is an autonomous and proactive piece of software. This means that they are capable of working within a cell and doing a particular task. Each agent should be able to describe itself in terms of what it does and what it can offer to other agents and it should be able to interact with those agents. Finally agents should be able to move between cells if it is necessary to do work more effectively.

In this approach, the behaviour of the agent only needs to be encoded in the software of the agent. As the agent requires particular resources it can query other agents to find those resources and then communicate with those agents to receive necessary information or relocate itself to another cell to access the resources that it needs. Agents need to share two ontologies, one describing the nature and capabilities of the agents themselves and a second describing the domain in which they are working.

This generally encourages individual agents which have fewer capabilities, but more agents in the system. This allows the system to continue to exhibit a diverse set of behaviours for a particular activity, but increases the flexibility in how the system is organized. For example in one simple application, three agents work together to create an RFID jukebox. In this system principals can throw poker chips marked with the name of a song onto a sensor. An agent attached to the sensor (in a cell with the RFID sensor shadow) can then detect the chip and send a message to a Database agent with the chip-id. With the chip-id the database agent can look up the song the principal wanted to play and then send that information to a DJ agent which is able to play the song.

Compared to a monolithic system, using the HIVE methodology with many agents, each
specialized for a particular task, it was significantly easier for developers to create the three agents (as the authors evaluated). It increases the flexibility of the system by encouraging reuse of agents. For example, in another test application of wearable computing, agents were able to determine where a particular principal was and ask a DJ agent to play his “theme song” as he entered a room (and even conceivably for a DJ agent to ‘follow’ the principal from room to room).

An approach such as Hive focused on a principal is an effective way to approach creating personal agent systems. It allows the principal to construct the systems they want from agents that have a common scheme for communication, but different internals. In the end the agents only need this common communication interface. Hive does not provide a common communication interface, but it does provide a good model for overall system construction.

With an understanding of the context it is now possible to describe a general, formal description of a framework for constructing a personal agent system, as well as to describe some philosophical guidelines which can help guide developers in creating practical personal agent systems for principals. These are presented in the following two chapters. Following those two chapters are three chapters that contain descriptions of a reference implementation for the framework, then three personal agent systems implemented using the reference implementation and finally evaluations of the applicability of those three systems.
Chapter 4

Definitions of the Task and Workslip Scheme for Personal Agent Systems

The Task and Workslip Scheme for Personal Agent Systems describes the relationship of the task to be solved to the workslip, and the agents which can do the steps of the task. Each agent represents a transformation to the workslip, with some workslip fields being read and others being updated. Each agent works on one step of a task, and a task is resolved with a personal agent system (built out of these agents) when it has worked on every necessary step.

This chapter provides a formal definition for each necessary concept of the scheme. This includes the Principal, the Task, the Workslip, the Communication Structure, the Environment, the Agent and the Personal Agent Systems. These concepts provide the foundation for the operation of a personal agent system assisting a principal to complete a task. In the following chapter, philosophical guidelines are presented to help developers create helpful agents and helpful personal agent systems using these concepts.
4.1 Principal

Personal agent systems must be viewed in the context of a person, the principal, who has tasks that need to be done in a digital environment.

Definition 1. A principal, $\mathcal{P}$, is the person who creates and uses the Personal Agent System.

Personal agent systems surround this principal with a “galaxy” of personal agents, which are able to cooperate to work on these tasks. Some agents may work on many tasks and some may be specialized for specific tasks. When a task needs to be done, agents on parts of the task, cooperating to complete the whole task.

4.2 Task, Task Steps and Task Instances

Tasks are any activity that the principal would like to do in a digital environment. These include interacting with software, transferring or transforming data, taking observations from the outside world (outside of the principal’s computational space) or sending messages outside. Tasks describe a very wide range of activities, but the focus here is on digital tasks which agents can undertake for a principal.

Definition 2. A task, $\mathcal{T}_a$, describes a piece of work to be done or undertaken for a principal in a digital environment of a principal.

Tasks can be broken up into task steps which are the different pieces of work that need to be done for a task to be done. For example, if a task is to search for news relating to a particular concept, then the steps of that task include checking each of the principal’s
preferred different news sources and evaluating the news from those sources to see if it relates. The next step might be determining if a news item is “important enough” and then delivering it to where the principal wants. Task steps might be considered tasks of their own, or a task might only have one step. Task steps may also be shared between different tasks.

Definition 3. A task, $T_a$, consists of task steps $(t_{a1},...,t_{an})$, where $n \geq 1$, that must be completed for $T_a$ to be complete.

Each task step $t_{ai}$ of $T_a$ is a task unto itself and may consist of one or more task steps.

It is necessary to distinguish between the task as an abstract set of work that can be done, and as distinct set of work being done at a particular point in time. The task instance describes the particular work being done at a particular point in time. For example, “get the news” is a task where as “get today’s news at 6pm” is a task instance.

Definition 4. The task instance $Ins(T_a, t)$ is the instance of task $T_a$ that begins at time $t$.

### 4.3 Workslips and Master Workslips

Workslips encode the information that personal agent systems need to carry out a task. For each task there will be a master workslip which describes all of the information needed for the task. For each task instance there is a workslip, which instantiates the master workslip. This workslip includes the necessary information for that task instance. This information is in the form of workslip fields and a workslip consists of a set of these fields where each
field includes information that the agents in the personal agent system must gather from the outside world or generate. A new workslip is created each time work begins on a task and agents pass the workslip from agent to agent, allowing each agent to contribute the necessary information and to use the information already gathered to do their work.

The workslip may be viewed as being similar to a blackboard for agent communication in that they provide a structured space where in agents may share information. However they differ in that there is a single workslip per task instance and that all information provided in the workslip is in the form of added knowledge with regards to the problem which another agent can act on, by prior agreement. Workslips pass from agent to agent in a flow of interest and capability in the way a job may be passed along an assembly line from technician to technician.

Workslip fields are made up of two parts, the field information and the value. The field information is used to describe the field and includes a name, as well as information about the type of information that the field can include, the potential values the field can contain, and a human-readable description.

The name is used to identify the field and must be unique among all of the fields for all of the workslips for all of the tasks for which the principal has personal agent systems. The type describes the type of data that the field will contain (boolean, integer, real number, string, identifier). This does not need to correspond to any programming language’s type definitions, but should be meaningful to the principal and agent developers. The potential values of the field allow a restriction on the type to ensure that the information contained in the field is meaningful (for example, this field should only contain even, positive integers). Personal Agents can use these restrictions to ensure that the workslip is correct at any point in time. The human-readable description allows the principal to understand the purpose of the field, the types of information it might contain and how it should be used.
4.3. WORKSLIPS AND MASTER WORKSLIPS

The value of the field is provided by the personal agents based on the information gathered or generated, or the value is undefined if no agent has given the field a value yet. As agents work, they gather and generate information and update values of the fields.

**Definition 5.** A *workslip field*, \( f \), is a pair of the form

\[
    f = (f.info, f.value)
\]

where \( f.info \) is a tuple, containing information that describes and identifies the field,

\[
    f.info = (\text{name}, \text{type}, \text{potential\_value}, \text{description})
\]

where:

- **name**: is a string that uniquely identifies the workslip field for the principal.
- **type**: is a data type definition \( D_{\text{name}} \) that describes the type of the \( value \).
- **potential\_value**: describes restrictions \( r \) on \( D_{\text{name}} \) about what type of information the \( f.value \) is allowed to have.
- **description**: is a string that provides a human-readable description of the field, including the **type** and **potential\_value** information about the \( f.value \).

and \( f.value \) is an element of the set of acceptable data values,

\[
    f.value \in A_{\text{value\_name}} = \{\bot\} \cup \{s \in D_{\text{name}}|r(s) = \text{true}\}
\]

The master workslip provides a list of all of the fields that may be necessary for personal agent systems to work on a given task.
4.3. WORKSLIPS AND MASTER WORKSLIPS

**Definition 6.** A *master workslip* $\mathcal{W}(T_{a})$, for a task, $T_{a}$, is a set of workslip field information tuples that describe the information potentially required to complete task $T_{a}$

$$\mathcal{W}(T_{a}) = \{ f_{1}.info, ..., f_{m}.info \}.$$ 

For each task instance, the workslip describes the information produced by the personal agents. As agents work, they update the values of the workslip fields to reflect the results of their work.

**Definition 7.** For a task instance $\text{Ins}(T_{a}, t)$ of task $T_{a}$, the *workslip* $w(\text{Ins}(T_{a}, t))$ is the set of workslip fields

$$w(\text{Ins}(T_{a}, t)) = \{ f_{1}, ..., f_{m} \}.$$ 

such that each workslip field is a tuple with a value for each work slip field information tuple in $\mathcal{W}(T_{a})$ where

$$\text{if } f_{i}.info \in \mathcal{W}(T_{a}), \text{ then } f_{i} \in w(\text{Ins}(T_{a}, t)).$$

Each workslip represents an element of the *Potential Workslip Set*. The potential workslip set represents all possible combinations of acceptable data values across all fields in a master workslip. Each workslip is therefore a particular element of that set and each agent update to a workslip is a transition from one element to another.

**Definition 8.** For a given master workslip $\mathcal{W}(T_{a})$, the *potential workslip set* $\mathcal{W}_{\text{set}}(\mathcal{W}(T_{a}))$ is the set of all possible workslips that use the fields indicated in the master workslip, such that each workslip $w(\text{Ins}(T_{a}, t)) \in \mathcal{W}_{\text{set}}(\mathcal{W}(T_{a}))$ represents a unique arrangement of values over the workslip fields.
4.4 Communication Structure

The workslip communication structure provides a common interface for all agents to receive and send information in the form of workslips. The communication structure can be viewed as an abstract structure which each personal agent is able to access, receive information from and update. The communication structure is an array of data spaces, each corresponding to a workslip field for the task being undertaken.

**Definition 9.** For each workslip, \( w(\text{Ins}(T_a,t)) \), there exists a communication structure \( \text{comm}(w(\text{Ins}(T_a,t))) \).

The communication structure consists of data areas

\[
\text{comm}(w(\text{Ins}(T_a,t))) \subseteq D_1 \times \cdots \times D_m
\]

such that there exists one data area \( D_i \) for each field \( f_i \) in the workslip.

As agents work they update the communication structure with new values they generate in the workslips they produce. At a particular point in time the communication structure will contain the set of values that have most recently been provided to it by all of the agents working on the task.

In this instance the symbol \( D_i \) is used to refer both to the data area for a particular workslip field, and to the data type for that area.

**Definition 10.** At time \( k \) the value \( \text{value}(k, \text{comm}(w(\text{Ins}(T_a,t)))) \) is the set of values that an agent is able to read from the communication structure. The value is defined such that,

\[
\text{value}(k, \text{comm}(w(\text{Ins}(T_a,t)))) = (d_1, \ldots d_m) \text{ where } d_i \in D_i
\]

for each \( D_i \) of \( \text{comm}(w(\text{Ins}(T_a,t))) \).
When the personal agent system begins working, the communication structure takes an initial value where all fields begin unset. As the system operates they will be updated to acceptable values for each field (as necessary).

**Definition 11.** When it is initialized at time \( t \), the value of each
\[
d_i \in \text{value}(t, \text{comm}(w(\text{Ins}(\mathcal{T}a, t))))
\]
is defined to be \( \bot \).

For any data value in the communication structure agents will need to know when it was produced to determine its usefulness and whether it should be overwritten. The *update time* provides that value for the agents.

**Definition 12.** The function \( \text{time}(d_i) \) returns the *time* at which the value \( d_i \) within \( \text{comm}(w(\text{Ins}(\mathcal{T}a, t))) \) was updated most recently.

### 4.5 Environment

Personal agent systems must operate in the real world. The *Environment* describes the relationships of the real world to a principal’s personal agent systems.

**Definition 13.** For each instance \( \text{Ins}(\mathcal{T}a, t) \) of task \( \mathcal{T}a \), the environment \( \mathcal{E}(\text{Ins}(\mathcal{T}a, t)) \) describes the potential *environmental states*
\[
\mathcal{E}(\text{Ins}(\mathcal{T}a, t)) = \{\text{es}_1(\text{Ins}(\mathcal{T}a, t)), \ldots, \text{es}_k(\text{Ins}(\mathcal{T}a, t)), \ldots\}
\]
that the environment for a personal agent system for \( \text{Ins}(\mathcal{T}a, t) \) could take.

Each particular configuration of the environment is an *environmental state* and is described with respect to the agents’ assignment to computation resources and the values of the outside world that the agents are aware of but do not directly control.
**Definition 14.** The environmental state \( es_k(\text{Ins}(Ta, t)) \in E(\text{Ins}(Ta, t)) \) describes a configuration of the environment. Each environment state may be expressed as a pair:

\[
es_k(\text{Ins}(Ta, t)) = (\text{compute}_k(\text{Ins}(Ta, t)), \text{world}_k(\text{Ins}(Ta, t)))
\]

where:

- \( \text{compute}_k(\text{Ins}(Ta, t)) \): describes the *computational resources* available to execute personal agents and the arrangement of personal agents within this computational space.

- \( \text{world}_k(\text{Ins}(Ta, t)) \): describes the current state of the *information about the world* the principal’s personal agents can perceive.

Agents are described in detail later in this chapter, but for now it is beneficial to be able to describe where they are running. Agents are software and as such need hosts (computation resources, such as processors, memory, secondary storage, etc) to be executed on. For each personal agent system, the principal will require a set of these agents.

**Definition 15.** The set \( \text{Agents} = \{A_{g1}, ..., A_{gp}\} \) defines the set of all agents that may be used to solve any task step \( ta_i \in Ta \).

Additionally, the principal needs to have a set (at least one) of hosts which are capable of executing the agents.

**Definition 16.** The set \( \text{Hosts} = \{H_1, ..., H_q\} \) defines the set of *hosts* which represent the computation resources (available computers, physical and virtual) available to the principal to execute agents.
At any point in time as the personal agent system runs agents must be assigned to the resources needed to run. The set $Assignment_k$ defines the current assignment of agents to hosts. Generally, an agent would be assigned to only one host (and several agents could be assigned to a host at the same time), however it is worth noting that a personal agent may itself be a multi agent system and may have components assigned to several hosts.

**Definition 17.** For the set of agents, $Agents$, and the set of hosts, $Hosts$, the computational space at time $k$ is defined:

$$compute_k(Ins(Ta,t)) = (Hosts_k, Agents_k, Assignment_k)$$

- $Hosts_k \subseteq Hosts$, is the subset of hosts available at time $k$.
- $Agents_k \subseteq Agents$, is the subset of agents running at time $k$.
- $Assignment_k$ is the set of pairs assigning agents to hosts, such that
  $$Assignment_k = \{(H, Ag) | (H \in Hosts_k \text{ and } Ag \in Agents_k)\}$$ at time $k$.

The *world* describes the information about the world that the agents cannot manipulate. This may include information from other services around the internet or from devices attached to the hosts that the agents could interrogate. Each potential piece of information that an agent might need is described as a data area. As an example, a piece of information might include a weather forecast from a weather service, or a temperature reading for a sensor attached to the host, or a sentiment evaluation provided by the principal on their phone.
Definition 18. For each task instance $\text{Ins}(T_a, t)$, there exists a world information structure $\text{world}(\text{Ins}(T_a, t))$. The structure consists of data areas

$$\text{world}(\text{Ins}(T_a, t)) \subseteq E_1 \times \ldots \times E_r$$

such that there exists one data area $E_i$ for each piece of information about the world which is controlled by a factor outside of the personal agent system.

At a particular time, each data area takes on a specific value that agents can then observe.

Definition 19. At time $k$ the world value is the set of current values for each data area $E_i$ of $\text{world}(\text{Ins}(T_a, t))$ such that

$$\text{world}_k(\text{Ins}(T_a, t)) = (e_1, \ldots, e_r) \text{ where }$$

$$e_i \in E_i \text{ for each } E_i \in \text{world}(\text{Ins}(T_a, t))$$
4.6 Agent Definitions and Agent Instances

An agent is a piece of software that must have a host to run. It runs within an environment and with access to information about an environment and more importantly with access to information about the actions other agents have taken through the workslip for the task instance. Each agent needs to know which fields in the workslip need to have values updated for it to work and which fields it will update when it is finished. There are also usually conditions on which values the input fields must have before the agent can act. The actual action of the agent is governed by two functions, its activation function and its action function.

The activation function checks the current workslip to see if at least one of the conditions are met, in which case the agent can then run its action function which takes the workslip and the current state of the world into consideration, does whatever work it may do and then outputs the result as an updated workslip.
Definition 20. An agent $A_g$ is defined by the tuple

$$A_g: (\text{in}, \text{condition}, \text{out}, \mathcal{E}(\text{Ins}(T_a, t)), \text{comm}(w(\text{Ins}(T_a, t))), activation_{A_g}, action_{A_g}).$$

where:

- $\text{in} = \{f_1, ..., f_i, ..., f_x\}$
  
  \[ f_i \in w(\text{Ins}(T_a, t)) \text{ and } \text{reads}(A_g, f_i) \text{ is } true. \]

- $\text{condition} = \{\text{condition}_1, ..., \text{condition}_i, ..., \text{condition}_y\}$ is a set of predicates over the values of fields in $\text{in}$, which define the possible conditions under which the agent should work on a workslip with the values of $\text{in}$.

- $\text{out} = \{f_1, ..., f_i, ..., f_z\}$
  
  \[ f_i \in \text{workslip} \text{ and } \text{updates}(A_g, f_i) \text{ is } true. \]

- $activation_{A_g}$ is a function
  
  $$activation_{A_g} : W_{set}(W(T_a)) \rightarrow \{yes, no\}$$

  where $activation_{A_g} = yes$ at time $k$ if for at least one $condition_i \in condition$, $condition_i = true$.

- $action_{A_g}$ is a function
  
  $$action_{A_g} : (W_{set}(W(T_a)) \times \mathcal{E}(\text{Ins}(T_a, t))) \rightarrow W_{set}(W(T_a))$$

  such that the fields of $\text{out}$ are possibly updated to new values.

The $\text{reads}$ function is used to determine which fields are important for which agents to work. An agent needs to read a field if it needs information from it to make a decision about
whether it should work now or if it needs more (or different) information to do its work.

**Definition 21.** The function \( \text{reads}(Ag, f_i) \) returns \textit{true} if the agent \( Ag \) requires data from field \( f_i \) in order to operate and \textit{false} if it does not.

The \textit{updates} function is used to determine which fields are updated by an agent. An agent updates the field if it can produce a new value for the field after it has worked.

**Definition 22.** The function \( \text{updates}(Ag, f_i) \) returns \textit{true} if the agent \( Ag \) may change the \( f_i.value \) of the workslip field \( f_i \) after it has run and \textit{false} if it will not.

Agents work by taking a snapshot of the communication structure in the form of a workslip. They are then able to use that workslip to test if they should run at the current time, using the activation function. If it is time to work then the agents take the workslip and the environment and use it to do whatever work they need to do. In doing so, they produce a new workslip, with the values they update potentially changed. An agent’s work may also have a side effect, such as sending a message.

Once the agent has created the new workslip it then updates the communication structure filling in the places where it has information, or new information to share with other agents. Once the update is complete, the agent can wait to see if it needs to activate again.
Definition 23. At time $k_1$, an agent $A_g$ is able to operate provided it is assigned a host such that there exists an $(\mathcal{H}, A_g) \in \text{Assignment}_{k_1}$ for some host $\mathcal{H}$, and that it has access to the communication structure $\text{comm}(w(\text{Ins}(\mathcal{T} a, t)))$ and the world $\text{world}(\text{Ins}(\mathcal{T} a, t))$.

The agent takes the value of the communication structure and the world,

- $w_{k_1}(\text{Ins}(\mathcal{T} a, t) = \text{value}(k_1, \text{comm}(w(\text{Ins}(\mathcal{T} a, t))))$ and
- $\text{world}_{k_1}(\text{Ins}(\mathcal{T} a, t))$

It then executes the function $\text{activation}_{A_g}(w_{k_1}(\text{Ins}(\mathcal{T} a, t)))$. If the function returns $\text{no}$, it halts and waits for a later time.

If the function returns $\text{yes}$ then the agent executes the function $\text{action}_{A_g}(w_{k_1}(\text{Ins}(\mathcal{T} a, t)), \text{world}_{k_1}(\text{Ins}(\mathcal{T} a, t)))$. This produces the workslip $w_{k_2}(\text{Ins}(\mathcal{T} a, t))$ for some time $k_2$ after $k_1$.

The agent then updates the communication structure with the values in $w_{k_2}(\text{Ins}(\mathcal{T} a, t))$.

For all areas $D_i$ of $\text{comm}(w(\text{Ins}(\mathcal{T} a, t)))$ and for all fields $f_i \in w_{k_2}(\text{Ins}(\mathcal{T} a, t))$, the agent sets the value $d_i \leftarrow f_i.\text{value}$ if $d_i \neq f_i.\text{value}$ and $\text{time}(D_i)$ is at or before $k_1$.

The agent then halts and waits for the next time it can run.

4.7 Personal Agent Systems

To work on a task, the principal assembles a set of agents, which become the personal agent system for that task. For a personal agent system to be complete, there must be at least one agent that can update each field of the master workslip for that task. In a particular instance, a field might not be necessary, so not every agent may need to run for every instance
of the task.

In the personal agent system, several agents may update the same field. The default mechanism in this situation is that the most recent update to the workslip is accepted. If two agents are running which update the same field, the later update will be read by any following agent. If a different mechanism is needed to handle this conflict differently, it must be implemented as an agent, or at least as part of another agent.

**Definition 24.** For a given task instance $\text{Ins}(T_a,)$, with the master workslip $W(T_a)$, a personal agent system is the set of agents $\text{Agents}$ which is able to perform all of the task steps necessary to complete the task instance.

For each task step $t_{a_i} \in T_a$ there must be at least one agent $A_g \in \text{Agents}$ that is capable of completing the step. Further, for every workslip field $f_j \in w(\text{Ins}(T_a, t))$ there must be an agent $A_g \in \text{Agents}$ with $f_j \in \text{out}$.

A personal agent system has finished its work when it is able to produce values in the workslip that meet the goal of the principal. There may be several such goals, but once one of them has been reached the agents can stop working.

**Definition 25.** The Goal Set $G(\text{Ins}(T_a, t))$ is a subset of the potential workslip set $W_{\text{set}}(W(T_a))$, such that for each workslip

$$w(\text{Ins}(T_a, t)) \in G(\text{Ins}(T_a, t)) \subseteq W_{\text{set}}(W(T_a))$$

$w(\text{Ins}(T_a, t))$ contains values in a subset of its fields which do not require further translation by any agent.

It may prove useful to track the activity of a personal agent system, for auditing, debugging, performance or simply informational reasons. Fortunately the workslip structure and the communication structure make it simple to track where agents make updates. With
this information, as well as the logs from the agents providing justification for the decisions they made, a trace can be created showing the operation of a personal agent system for a particular search instance.

**Definition 26.** The tuple $update(w(\text{Ins}(Ta, t), Ag, t'))$ represents the updates made to workslip $w(\text{Ins}(Ta, t))$ by agent $Ag$ at time $t'$, some time after time $t$.

**Definition 27.** For the set of agents $Agents$ between times $t_{\text{start}}$ and $t_{\text{end}}$, the sequence of updates to workslip $w(\text{Ins}(Ta, t_{\text{start}}))$ form the trace $trace(w(\text{Ins}(Ta, t_{\text{start}})), t_{\text{end}})$.

These definitions provide the broad framework within which to discuss the function of personal agents and personal agent systems in the Task and Workslip Scheme. In the following chapter these concepts will be refined with philosophical guidelines to ensure that personal agent systems built with the scheme are as helpful as possible to principals.
Chapter 5

Philosophical Guidelines for the Design and Implementation of Personal Agent Systems using the Task and Workslip Scheme

The Task and Workslip Scheme for Personal Agent Systems is an approach that allows agents to cooperate to work on a task for a principal using the structure of a workslip to organize their communication and cooperation. The previous chapter described the definitions for tasks, workslips, agents, and personal agent systems. On their own, these definitions provide a broad outline of how a personal agent system can be designed, developed, deployed and used. This chapter provides some philosophical guidelines, to help guide the creation of these systems. These guidelines provide guidance on the selection of tasks, the design and implementation of agents, the structure of workslips and the construction and use of personal agent systems.
There are some broad questions that should be addressed before looking at the specific guidelines: How would a principal expect to use a personal agent system to complete a task? How would a personal agent system operate? Where do the agents a principal uses for their personal agent system come from? Why would a principal choose to use a personal agent system? How specific should a personal agent system be?

The Task and Workslip Scheme is created with the expectation of a principal who has a handful of computing devices. These would include a small number of desktop computers, laptops, tablets or phones. These might also include access to “heavier” computing devices such as a server or a cloud service or “lighter” computing devices such as a smart watch or household devices with computational power such as an Internet of Things Hub or a Smart Television. A further expectation is that software running on any of these devices is able to communicate with software running on any other device.

A principal with a task would expect to be able to run several agents within this environment of devices. In the abstract, an agent could be run anywhere, but generally most agents will be run on the more general platforms such as a desktop computer and only specialized agents will be run on more esoteric platforms. A decision agent might run on the principal’s desktop, whereas there might be an agent dedicated to determining what items are left in the refrigerator running there. Equally if an agent needs a lot of computing power, it might run on a server instead, for example when it needs to generate a complex plan.

A principal would expect to create a new personal agent system by asking a management interface to find agents that can provide particular answers (in the form of fields of a workslip). The management interface would keep a list of agents and know which agents were able to provide the given fields. The principal can then work through the system with the management interface, adding agents that provide information needed by other agents until paths exist that allow information to flow from source agents to the agents providing
the answers to the principal.

Consider the alarm clock example from Chapter 2. Alice wants her alarm clock time to change with the weather. To update her clock, Alice needs an agent that updates the alarm time and sends it to her clock. To determine how much the alarm time should change, she needs an agent that can evaluate the weather and communicate it to the time changing agent. To evaluate the weather, she needs an agent that can query the local forecast and digest the weather for the evaluation agent.

This is not to say that the principal creates paths of information flow. Rather agents need to be made available to supply information as needed. Agents themselves do not need to know which other agents will receive the information they produce, nor do they need to know where the information they are working with comes from. In cases where particular conditions need to be handled, such as responding to the failure of one agent by invoking a more costly agent, the principal can use specialized agents to route information in specific cases, dividing information where necessary and then recombining it according to special rules.

There may be several different evaluation agents which can send updates to the control agent, Alice does not need to specify which agents should communicate in which way, only that the agents should be running and that the control agent should consider the different inputs in the way she would like the information considered. Conversely, if Alice is concerned that her preferred weather forecasting service might fail, she might add a management agent that checks to see if the agent for the preferred service is working and that invokes an alternative agent that checks the weather from her less preferred source.

In another situation Alice might find that the control agent only responds to the information provided by the last evaluation agent to complete execution (overwriting the information in the workslip provided by the agents that had already executed). This might be the be-
haviour she wants, or she might need a special agent to combine the information into a digest that can be send to the control agent.

If principals are not expected to be software developers, where do the agents come from? It is expected that every principal has access to a library of agents, but how does that library get filled? Ideally, libraries will be filled from agent stores which will be populated by agents created by agent developers. Principals will access the stores with their management interfaces when their libraries do not contain agents that can fulfill the need of the system they are creating. Agents might be purchased from a developer. Once retrieved, agents can be added to the principal’s library and then deployed to run in their environment.

This aspect of the Task and Workslip Scheme for Personal Agent Systems is simultaneously one of its most important benefits, but also one of the aspects where the potential to fail is the greatest. As discussed, personal agent systems are expected to be personal. However it may be difficult for a principal to ascertain that an agent is working in their best interest. If Store X has provided the agent that evaluates which store provides the best cost on an item, what guarantee does the principal have that the agent does not simply list Store X as the best place to purchase any item the store carries. In less egregious cases, how can a principal tell that an agent is bug free and working correctly?

This is left as an open question. Automated checking of the behaviour of agents is important, but is a weighty topic on its own. Ideally communities will come forward to bridge this gap, providing reviews, recommendations and certification. This will provide an environment in which the principal is able to find the right agent for the job and be assured that the agent will behave the way they expect it to [ASH+13].

Community is also the ideal answer to the question, why would developers choose to create agents? Creating an agent should be rewarding for the creator, either financially, or from personal satisfaction (assuming these rewards are not detrimental to the principals).
A developer should also expect to be able to benefit from having a community of other developers creating other agents. A healthy community of developers working together would provide the best possible agents to a healthy community of principals.

Beyond the practical questions above, there are two concepts that will be needed to discuss the philosophical guidelines. The first is, what is “worth it” to the principal, and the second is when is a task step or an agent small (or simple) “enough”.

When discussing the value of worth, the question becomes: will the principal benefit from spending their money or their time on this aspect of a personal agent system? Consider Alice from Chapter 2 again, creating a personal agent system for grocery list management. One task step of that task is that she needs a way to calculate the common price for each item on the shopping list. For this task step, she will have to find an agent able to work on that step, and pay the developer, or she will have to create her own agent. This is a critical task step in the task for her, so it is worth the commitment of time or money.

In other situations it may be worth the cost if the task step is one that appears in many of the principal’s tasks, but if the step is rare or can be avoided then one principal may decide that it is not worth the cost, whereas another might decide that it is. Worth is determined by each principal for their own purposes, in their own time. The guidelines depend on the principal finding the value of work with the system within their own lives.

Similarly to worth, the concept of sufficiently (or enough) will surface several times during the discussion of the guidelines. Here, the question is, does the system adhere closely enough to the general philosophy of the scheme to reap the benefits of the scheme without creating undue complication, complexity or work for the principal.

Consider Alice again, this time with the aware alarm clock. How many task steps are there in the task? Are there two steps? Four steps? Eight steps? A thousand steps? All of these divisions might be “correct”, but when has the task been divided enough?
To benefit from the scheme, more steps are probably beneficial, both to allow more specific (and smaller) agents and to allow steps to be shared between tasks. Alice could have a single agent that retrieves the weather report, processes it, and checks how the weather meets her rules for changing the alarm time. This would be allowed within the scheme, but the agent would be more complex, harder to create, and would require more maintenance over its lifetime.

It may be better to divide the complex agent into several simpler agents; one agent to retrieve the weather, one agent to evaluate the weather, one agent to compare the weather to Alice’s rules. If Alice is later creating another system that requires knowledge about the weather, she already has an agent available that does the work the way she likes. She can also add or replace agents without interfering with the work of the other agents.

That being said, there can be too much division as well. Creating too many divisions requires the creation of too many agents or possibly the task might be divided into task steps so specific that they could only be needed for the one task, or they would simply be trivial. For example, Alice could have one agent to retrieve each value of a weather report, or an agent that is responsible for doing addition between two numbers or concatenating strings. In each of these cases, it seems that the creation and maintenance of the agents would be too high an overhead compared to the benefits of the system.

As with worth, it is the principal’s responsibility to determine when enough is enough. When is a problem divided well enough? When is an agent simple enough? When are enough cases covered by the system? All aspects of personal agent systems using the scheme can be stretched to absurd cases, but it is better to limit the cases of interest to situations where the solutions are sufficient for the principal.

The remainder of this chapter presents guidelines for the use and design of a personal agent system. These guidelines are grouped into four groups: the selection of tasks, the
5.1 WELL-SUITED TASKS

design and implementation of agents, the structure of workslips and the construction and use of personal agent systems.

5.1 Tasks That are Well-Suited for the Task and Work-slip Scheme

There are a broad range of activities that fit the definition of a task within the scheme. However, not every conceivable task is readily implemented by a personal agent system. As a simple example, consider if Alice wanted to build a tree house. This would certainly fit within the definition of a task, it describes a piece of work to be done and it can be divided into task steps.

However, building a tree house is poorly suited to personal agent systems. It needs to be done in the real world, which is difficult for electronic agents to work in. It has a wide variety of potential inputs and outcomes which makes it difficult to define. Finally Alice is likely to only need a few tree houses over the course of her life, so automation has very little benefit in the long run.

Instead, well-suited tasks should be expected to demonstrate three particular attributes; the task must be repeated (done often enough that a system is useful), regular (sufficiently specific that the outcomes can be described), and informatic (of the electronic realm). The restrictions reduce the potential set of tasks a principal can set a personal agent system to do, but help make it easier to design, plan and implement a system.
5.1. WELL-SUITED TASKS

5.1.1 Repeated Tasks

A well-suited task must need to be accomplished with sufficient frequency that it is worth the principal's time to construct the personal agent system to accomplish it. Information Technology professionals are frequently faced with the question "At what point should I write a script, rather than process this data manually?" Is it beneficial to spend the time writing a script to do a job, if a job is done once a year and takes ten hours each time? If doing the job with the script takes nine hours, it may not be beneficial. If using the script reduces the job to fifteen minutes, then it may be beneficial. If the script improves the principal's state of mind, then it might also be, even if the actual time benefits cannot be demonstrated.

For personal agent systems the question is essentially the same, "Is the task sufficiently repeated that it makes sense to create a personal agent system to do the work?"

**Definition 28.** A task, $T_a$ is repeated if it occurs sufficiently frequently that a principal $P$ believes it is more efficient to create a personal agent system to address the task, rather than to do the task without the system.

A task that only occurs once is not a suitable task for a system using the scheme. In most circumstances the work of creating the personal agent system would not be outweighed by the work of completing the task. Furthermore, it is likely that the task will not be understood well enough by the principal to create the personal agent system in the first place. Alice needed to understand how Bob did each of the tasks she gave him before she was able to create systems to replace him. In execution of the task it is also likely that the personal agent system would not produce any better results than if the principal did the work themselves, or if they programmed a script in some other environment.
There is also an upper limit on how repeated a task can be for a personal agent system to be useful. A task which occurs very frequently may be better served by using dedicated software. Personal agent systems using the scheme are intended to fill an intermediate role in people’s lives, allowing them access to systems that behave like custom software, but which they can construct themselves. They will not be the most efficient software in the vast majority of cases, however the benefit in flexibility and personalization will outweigh the deficit of efficiency.

Consider if Alice were to start her own store. A personal agent system might prove useful to help manage employee schedules (although personal agent systems with multiple principals is outside the scope of this scheme), but it would not be the correct choice for inventory management or the point of sale system. In those cases Alice would want to look for, or create, dedicated software solutions.

Looking at the three scenarios from Chapter 2, each of the situations that Alice is trying to improve involves a repeated task. Of the three, the “Aware Shopping List” runs the least frequently, with agents running a few times a week, the “Aware Alarm Clock” runs more frequently with agents running a few times an hour, and the “Proactive System Operator” runs most frequently with agents needing to run nearly continuously to observe changes in the system.

All of these tasks are clearly repeated tasks, and in all of these tasks the benefit the systems provide is sufficient that it will balance the cost of creating the systems, especially if the necessary agents already exist. Of the three the “Proactive System Operator” raises the most questions about the appropriateness of a personal agent system (the other two being very well-suited in this regard), since software to monitor systems and networks exists and is well-integrated (to a greater or lesser extent) in most IT environments. The argument for Alice to respond to this scenario with a personal agent system, revolves around the fact
that her environment is relatively small, and that she is a programmer, who is comfortable creating new agents that are specific to her environment, and the way she would like it managed.

### 5.1.2 Regular Tasks

A regular task is one for which the outcomes can be defined or identified and written into a workslip.

**Definition 29.** A task, $T_a$, is regular if all of the instances of the task have a goal state $G(Ins(T_a, t))$, that can be identified in a workslip produced by a personal agent system working on task $T_a$.

A task such as determining if a document adheres to the specified format would be considered regular whereas determining if a document *looks nice* would not. This is because the outcomes of the first can be defined, at a first level as a simple boolean statement, but failure to meet the specified format could be codified and listed.

As with other aspects of the guideline definitions, the conception of the task depends on the principal. If Alice has a sufficiently codified definition of what “looks nice” then this means that an agent can evaluate it, and the task is regular. If “looks nice” is not codified, then it is difficult or impossible for an agent to evaluate it. Similarly, an agent should be able to “buy tickets to the next Star Wars movie as soon as they are available” (though that task may fail the guideline of being repeated) and “buy tickets to a movie for us to watch for date night tonight”. If there is codification of what the principal wants (usually in the form of an agent, that either gets information directly from the principal or learns it) then the task remains regular.
5.1. WELL-SUITED TASKS

The three scenarios from Chapter 2 each solve a regular task. Updating the alarm produces outcomes of a different alarm time or not, depending on Alice’s specifications. Managing the shopping list produces a set of lists for each of the stores Alice will visit. The System Monitor must be able to alert the operator if the network condition falls into an identified pattern. Each of these tasks has specific requirements that a principal can codify to allow agents to work.

5.1.3 Informatic Tasks

The final guideline of a task in order to work within the Task and Workslip Scheme is that it must be informatic. It must be constrained to computational systems. Agents in the Task and Workslip Scheme can use sensors to observe the outside world and can influence the outside world through sending messages, but they are software, and any task they work on must be a task that can be fulfilled by software.

Definition 30. A task \( T_a \) is informatic if it can be accomplished using software agents running in a computational system.

A task such as “find the current weather”, is informatic because an agent can consult an online service, or use sensors attached to a host to find information about the current weather. Tasks such as “plow that field” are not informatic, at least without the introduction of a robotic element to manage plowing.\(^1\) Tasks such as “enjoy this song” seem to be left to the realm of human emotion for now and are also not informatic.

Each of the scenarios from Chapter 2 describe informatic tasks. The system to update the alarm clock is informatic because it interacts with the weather services and the alarm system

\(^1\)Alternatively, the task “plan how to plow the field” would produce a plan for a principal and would be informatic.
by sending messages (a message can be sent to cause a physical, audible, effect into the non-informatic world). Similarly the system to manage the shopping list communicates with flyer services to receive information and communicate with the principal via sending messages (for example, OS notification or e-mail). The Proactive System Operator observes various pieces of information provided by the network either by message sending or by probing directly and again communicates with the principal via message.

5.2 The Design and Implementation of Agents

Agents are the actors in the personal agent system. They are responsible for completing all of the task steps, and communicating the results of their work through workslips. Agents are opaque to each other, each agent only interacts with the other agents through the fields that are changed in the workslip and can only communicate with other agents by changing fields in the workslip.

This opacity is important in that it allows personal agent systems built with the Task and Workslip Scheme to be decentralized. This allows the principal the flexibility to build their system to work in the way they would prefer. Rather than requiring a planning stage to organize when each agent will activate, it is only necessary for each agent to monitor the state of the workslip and activate if it is needed.

In order for agents to work in this fashion they need to follow three basic guidelines in their construction. They must be simple, reusable, and clearly and consistently documented. These guidelines will help agent developers create agents for principals that are easy to use, easy to fit together and easy to share.
5.2. THE DESIGN AND IMPLEMENTATION OF AGENTS

5.2.1 As Simple as Possible, but No Simpler

An agent should be the minimum amount of software required to complete a given task step. Generally, each agent should solve exactly one task step, unless there is an exception reason for it to do more than one.

**Definition 31.** An agent $A_g$ is *sufficiently simple* if it completes exactly one task step $ta_i$ of task $T_a$.

The simplicity of an agent, has a corollary in the simplicity of a task step. How does a principal know how simple a task step should be? Hopefully, in practice, a guide will grow out of the community over time as more principals share agents to work on similar tasks. A task step will be as simple as the agent created to work on it.

To look at this question, as a question of sufficiency, consider the three example scenarios that Alice experiences in Chapter 2. In the first scenario, Alice needs to get information from a weather service and convert it into a format that will be usable by later agents. What is the task step here?

Alice could have separate agents that get each element of the weather update, the current temperature, the current rainfall, the current visibility, and so on. This would produce significant overhead, both in terms of time executing, and in time communicating. Having each element of a weather forecast managed by its own agent seems as though the division would be too granular.

Alternately, Alice could have one agent that retrieved the weather and evaluated it all in one step. If Alice wanted to change the source of her forecast to one that used a different format, she would have to replace the entire agent. If she wanted to change how she evaluated the weather, she would also need to replace the entire agent. In this scenario, it seems as though the division is not sufficiently granular.
5.2. THE DESIGN AND IMPLEMENTATION OF AGENTS

Similarly when managing the shopping list, it makes sense for Alice to have agents which are able to retrieve sale information, agents which can manage the shopping list, agents which can combine the sale information with the shopping list, the result of which is eventually communicated back to the principal by an agent which sends e-mail.

In both of the first two scenarios, the work in each task step is relatively simple. Get information, do basic processing, test if conditions are met. For the Proactive System Operator, consider if Alice used a machine-learning-based intrusion detection system. This system would need to execute a complex algorithm, and would therefore require an agent with access to a large memory footprint and a powerful processor. While this might seem to be a “large” agent, it would be the simplest agent to do the necessary tasks. For the Task and Workslip Scheme, each task step, and therefore each agent, should be as simple as possible, but no simpler.

5.2.2 Reusable, Written Once and Used Everywhere

One of the great benefits to the simplicity expected of agents in the Task and Workslip Scheme, is that it makes reusability simple. Ideally a principal will have several different personal agent systems operating in their environment, and it would be useful for them to have agents that are able to work in different personal agent systems at the same time. Equally, an agent should be useful to as many principals as possible.

**Definition 32.** An agent $A_g$ is **reusable** if it can be used to work on task steps $ta_A$ and $ta_B$ in two different tasks $Ta_A$ and $Ta_B$ or for the same task for two different principals $Pa_A$ and $Pa_B$.

As described in Chapter 2 none of the scenarios demonstrate the need for reusable agents very well. However, with a little extension, these agents might be useful in other systems.
One aspect of the network monitor is that in some weather conditions, telecommunications equipment can function poorly. As such adding a group of agents to help operators be aware of the weather could be useful and the same agents could be used as in the weather retrieval portion of the alarm clock. Similarly, if the weather is bad, the shopping list manager might be able to push items on to the list for a closer store for Alice so that she will not have to drive as far in bad weather.

When designing agents, creators should consider potential other uses for their agents and try to design them in such a way that they can be easily used to work with other personal agent systems.

### 5.2.3 Clear in their requirements and production

Each agent needs to be well-documented for principals to be able to construct systems using the agents. When reading the document the principal should be able to clearly understand what information (which workslip fields) an agent needs to work, and what that agent updates after it works.

**Definition 33.** An agent $Ag$ is *clear in its requirements and production* if it can provide a list of fields it requires and fields it may update, which is readable both for a principal and for other agents.

This is an extension of the *reads* and *updates* functions described in the scheme in Chapter 4.6. When a principal inspects an agent, they should be able to identify the agent by its role in the transformation from one set of workslip fields to the next. This makes it possible for the principal to determine which agents best fit in the systems they are constructing. It also allows the principal’s management interface to describe the system’s potential behaviours, and which fields must still be produced in order for the system to work.
Additionally, this requirement helps to reinforce the requirement of simplicity. If an agent requires and updates dozens of workslip fields, then that is a sign that it may be too complex. This may not always be true, but it is a warning sign.

It is difficult to show this directly in the three scenarios from Chapter 2 because Alice’s primary solution to her problems is Bob. Chapter 6 outlines a reference implementation of the Task and Workslip Scheme, which illustrates an approach for agents to share their fields of interest.

### 5.3 The Structure of Workslips

The workslip is the data structure at the heart of every personal agent system. The work being done must eventually be described in terms of the workslip fields that are needed and being updated. Agents work by passing workslips along from one agent to the next. Agents are able to cooperate because they are able to describe their abilities and responsibilities in terms of the fields that make up a workslip.

There are many questions, however, about how a workslip should be designed and implemented. How do agents access workslips, and how do they share them? How do agents find the fields they need? What happens as one agent updates information that another one had previously updated? What fields describe a task or a task step?

To help guide the design and implementation of these workslips, there are three broad guidelines: They must be a portable, but inert, data structure that agents can share, they must be descriptive of their own creation and history, and they must be built out of fields that are specific and uniquely identifiable.
5.3. **THE STRUCTURE OF WORKSLIPS**

### 5.3.1 A Data Structure Sharable Between Agents

Agents must be able to access the information stored in the workslip. Because personal agent systems built using the Task and Workslip Scheme should be decentralized, workslips should be passed between interested agents rather than housed in a central location. The format agents use to interchange workslips should be as generic as possible, not relying on structure provided by any particular technology.

A workslip should not have algorithms embedded in it, rather agents should be responsible for manipulating workslips. If conflicts arise in the data stored in a workslip, agents must be responsible for resolving these conflicts. The default behaviour for an agent is that the most recent update is stored in the field.

If that is not the behaviour that the principal wants, then there needs to be a different implementation of the system. This might include changes within the agents themselves, for example, checking to see if the last update is more recent (the agent started running later, but finished faster) or if it is “better”, using information internal to the agent or hints from the principal (that agent checks a better weather source, so this agent should not overwrite it). Alternatively the principal may need to add extra agents to the system to explicitly manage conflict and ensure that the best information is available for agents later in the system.

### 5.3.2 A Data Structure Defined by Its History

Agents transform workslips. In some situations they will read a group of fields and update a different group. In other situations they may read and update the same fields. In either case the history of the workslip can be helpful to an agent to make the transformation.

In the first scenario, the history of the workslip is encoded in the fields. As an agent
updates a field, that information becomes available to any future agent which reads the workslip. In this way the changes to the workslip over time can be seen, because they are all available as fields. Later a principal, or another agent can use the information included in the workslip to reconstruct a trace of the decisions made by the agents, which allows the history of the workslip, and the task instance, to be understood.

In the second scenario, agents may need to repeatedly update the same field. In this case the surface of the workslip no longer maintains the full history of the workslip and the task instance. In these situations the information included in a workslip field can be included as the *current value* with *historical values*. These historical values can include, the value, the time at which that became the current value and the agent which updated the field to that value. There may be situations where an agent may also include the justification for the change it made.

This would allow other agents to inspect a field and determine what values it has held over time and why they were changed. This may be useful for their own decision making, or it may allow the agent to ensure that rather than simply the most recent value being stored in the workslip field, the most correct value is held instead.

This raises the question of whether keeping a history will inflate the size of the workslip too greatly. This will vary by the personal agent system, but generally any field in the workslip should only be updated by a small percentage of the agents in the system. As such, each field should have only a small handful of history entries, which will usually be a small percentage of the overall size of the implemented workslip.

For example, in the Aware Alarm Clock scenario, each field would be updated only once by each agent as the workslip flows through the system. In the Aware Shopping List scenario, each field for the best price of an item would be updated at most once per shop, a number of times likely to be less than a dozen. In the Proactive System Operator scenario, there
might be scenarios where agents need to vote on a course of action, in these situations a field history could be used to track the voting and have as many changes as agents participating.

Having history makes it simpler to construct complex personal agent systems. Among other benefits, it allows for exception handling across a system. Suppose for the alarm clock scenario, the weather source agent fails to retrieve a forecast, it would then update the fields it would normally fill with forecast information with error information instead. Then an error agent could read the workslip and activate a special case agent (possibly one that has a cost associated with it, or a less accurate one) which could run and update those fields. Once the fields have working values they could then be passed on to the control agent.

Similarly, if the principal wanted to break a job into parallel steps, an agent could be used to divide the workslip and then send it to several agents to work at the same time. Once those agents had completed their work a second agent could be available to combine those results. In this situation each agent would have a separate set of fields they update or a history would allow that second agent to perform conflict handling and produce the best possible result from the work of the agents running at the same time.

Finally, because all agents in a personal agent system are independent, it can be difficult to debug a personal agent system. Each agent can, and should, keep a log of its operations and especially decisions. Keeping the history in the workslip, either on the surface or as an addendum to the fields, makes it much easier to combine all of the necessary information into traces which can then be used to see the overall behaviour of the system, and hopefully enables the principal to understand why the agents acted the way they did and what, if any, changes need to be made for the future.
5.3. THE STRUCTURE OF WORKSLIPS

5.3.3 A Data Structure That Is Sufficiently Specific and Unique

The final consideration when constructing a workslip is how it should contain information. Which fields do agents need for a task, how do agents identify those fields, and how much information should be held in a particular field?

The answer to the last question is similar to the answers to the question about how much a task should be broken down into task steps. The information in each field should be as specific and immediately useable as possible. Each field should store a value in such a way that an agent reading it should have to do as little processing as possible to extract and use it.

As with the division of a task into task steps, it is possible to divide the workslip fields too finely. A weather forecast broken down into single words would be almost useless. However having the whole forecast contained in a single field requires that an agent do work to extract the forecast for a particular day or a particular value from a particular day.

Beyond that philosophy that each field should be as specific as possible, there should also be a bias towards keeping as much information as possible in a workslip. That insures that an agent is providing as much information as possible for future agents to use. So long as the agent is still only doing a single task step, the more information it can provide the more other agents can rely on it to do other task steps.

For the Aware Alarm Clock, it would be possible for the source agent to simply put the full text of the forecast into a field in the workslip. This means that later agents in the system would have to parse the information they need out of the one field. If the source agent parses that initial forecast into smaller pieces of information, (such as the current temperature, current wind condition, current alerts, etc), then it can put each of those pieces into their own workslip fields. The source agent could pass the raw weather report to
a parsing agent, which will parse the report into specific fields. This way if an agent is only looking to see if the current temperature has dropped below a particular point, it only has to take data from the current temperature field. In either case, it is worth retaining the full text of the forecast in the event that another agent has a use for that information.

The other question that needs to be considered is how agents are able to identify workslip fields. On first glance this seems like a simple question, as workslip fields have unique names. However the question remains, which unique names do the fields actually have? If the source agent will provide current temperature, will it be in “weather.current.temp”, “weather.temprature”, “weather.tempreture”, “weather.temperature.inC” or somewhere else entirely?

It is difficult to get developers to agree on what things should be named, and further, how it is impossible to produce ontologies that span any particular field let alone the breath of information different personal agent systems might use. The solution to this, however, lies in the nature of personal agent systems, the fact that they are personal.

The principal knows the problem they are implementing the system to handle. At creation time, they know which fields are read and which are updated, by querying the agents. Each field is self-describing both in type and potential value, and in the human readable description. As the systems are constructed, the principal is able to build a dictionary for the agents, noting fields that are identified by different names, and recording the different names the fields are known by. As agents run, they are able to consult this map, and produce concrete field names that they can then use to communicate with other agents.

Consider Alice developing her Aware Shopping List. She might have access to agents from another developer who was creating agents for a similar purpose, but was more interested in using them for economic analysis. They have an agent that calculates the cost per unit in a field “cost_per_unit” whereas Alice has the field “price_per_unit” in her agent that manages
5.4 The Use of Personal Agent Systems

At some level personal agent systems can be difficult to distinguish from “software.” After all, most software is designed to transform some input into some different output. One aspect that differentiates personal agent systems from software as a whole is its intended use. Personal agent systems are intended to be used by a principal in order to make their lives easier.

In Chapter 2, with each of the motivational scenarios, Alice is attempting to make a minor improvement to her life. In each of them, Alice is attempting to automate the task she wants done, and have it done the way she wants it done. Most software is written with the intention of helping people, but with a secondary motivation of profit for an organization. Some times that secondary motivation can be predominant, or possibly some organizations may have other motivations to provide the software they provide. In each of these cases however it means that the principal has less control over their computing than they might ideally have. Personal agent systems are intended to rebalance some of these potential inequalities.

These are wide ranging topics that have many factors and implications, this section focuses on three main philosophical guidelines which suggest ways personal agent systems can (and should) be used to maximum effect. These are, that each system should be personal, automatic, but guided, and efficient, but discreet.
5.4. THE USE OF PERSONAL AGENT SYSTEMS

5.4.1 Personal

It may seem redundant, but personal agent systems must be *personal*. This means that every action taken by the agents in a personal agent system need to be not only to the benefit of the principal, they need to be *explicitly* known and consented to by the principal.

There are many systems available that provide services like these ones. Many corporations provide electronic assistants that offer to help a principal, but generally these are not guaranteed to be working in the best interests of the principal. That is not to say that there is anything inherently unethical about these systems, but rather that they are outside the control of the principal.

This means that they may not serve a principal in several ways. In particular, they may require personal data that the principal does not wish to share, but must, in order to receive the service. They may also simply not work in the way the principal would like.

In the first case, a system may simply require this information to function properly. This may be a system which is intelligent or uses machine learning to provide a service. However, the principal will have to share information with the system’s controlling organization. This may expose the principal to risks of data exposure that they are not willing to take, even if the only use of the data is to complete work for the principal.

Alternately, organizations offering these services may wish to use this information for purposes other than the one for which it was originally needed. This may be for marketing, sales or many other purposes. While an organization might be clear in its terms of service and privacy policies, the principal will often have little choice other than to use the service and provide the information or opt out of using the service.

Additionally, the principal can often have very little control over these services. This leaves the principal at the mercy of the organization offering the service. This can mean
that function might not be as well-suited to the needs of the principal as the principal might prefer, or even that the service can change over time. It also leaves the principal exposed to the biases of the developers of the systems. This may mean something as innocuous as a system not understanding which league (or sport) the principal might want when they ask for “the football scores”, or it might extend to failing to recognize the principal’s voice due to an accent or as far as being unusable by the principal due to a disability or a medical condition.

The intention of personal agent systems is to allow the principal to complete tasks by bringing systems within their own sphere of computation. This gives the principal control over their systems and allows them to keep their personal data within the computers and storage they already control and trust. Ideally this does not come at the expense of lower quality software through these communities for personal agent development.

5.4.2 Automatic, But Guided

Personal agent systems should be able to work on their own, otherwise there is no benefit to the principal creating them. However, to be personal, their automatic work must be guided by the principal. In effect, the principal needs to guide the organization of the system (assembling the correct agents to work together), the set up of the system (how each agent is configured) and the actions of the system, reviewing the work of the agents and readjusting them as necessary.

This requires that the personal agent systems, and the agents themselves, be able to be receptive to this guidance. When a new agent is developed it should be as configurable as possible to allow principals to change its behaviour, and ideally should also be modifiable at a source-code level, if necessary. Ideally an agent will have preset configurations and
structures that make it simpler for principals who fit the “usual” operation of the system, but these should not be the limits of configurability.

Agents with a model of the principal (for example, an agent that holds a preference about alarm clock settings) need to be able to communicate with the principal and ask for refinement and correction when necessary. For example, if Alice’s system regularly gives her a suggestion that she thinks is wrong, there needs to be a mechanism in the system for Alice to correct the agent’s “misunderstanding” of the situation.

Principals using agents that use machine learning need to be careful in the context of a personal agent system. In these cases, the models of the principal and the world, held by the agent may change without the principal being aware of the changes. In these situations the agents need to provide updates that inform the principal about the changes. If possible, agents should be able to explain why these changes happened, and what the impacts will be. Additionally, agents that learn should provide a way for the principal to teach directly when the learning is not supporting their use of the system.

Consider Alice’s system for managing her shopping list. Usually the system is automatic. It runs a few times a week and uses its model of Alice (her shopping list and store preferences), and the world (the shops Alice would like to shop at) to produce a set of useful lists for her. Over time Alice or the world might change. One shop might be bought by another chain and change its catalogue system. In this case Alice might need a new agent to capture information from the new system. A shop might go out of business, and requiring Alice to reorganize her preferences. Alice herself might change, she might discover she needs a special diet, or have an experience at a store which would cause her not to return there.

In all of these cases, it needs to be simple for Alice to adjust the system to fit her preferences again. Agents need to accept guidance, for example if Alice asks the system to stop showing her offers on meat, then the agents need to quickly accept this change.
Similarly, when an agent needs to be removed or added to the system, to better support the behaviour that the principal wants, it needs to be simple for the system to accommodate those changes and continue working.

### 5.4.3 Efficient, But Discreet

A principal should be able to expect not only that their system will interrupt them with questions only when necessary, but they should also expect that their personal agent systems will not interfere with their other work. A principal’s computational sphere will include several different types of computer, such as smartphones, tablets, desktops and laptops, and possibly some “heavier” computing devices such as server or cloud resources.

Agents in a principal’s personal agent systems will share computational resources with the other software that the principal is using. If an agent is executing on the principal’s laptop and the principal is trying to do something computationally intensive, it does not benefit the principal if the agent prevents the principal’s other work from running.

At the same time, the principal would like the results of their personal agent system in a timely fashion. Therefore agents should run at a high priority when they are not interfering with the principal’s other work. As such, agents in a personal agent system need to work in the background as quietly and as efficiently as possible.

This ties back into the the concept of finding appropriate tasks for personal agent systems. Well constructed agents working on a human time-frame task should be able to work as quickly and as often as the principal needs, without producing an undue burden on the principals computational resources.

One example of how this balance needs to be maintained can be seen in Alice’s Proactive System Operator. The agents need to be able to quickly respond to problems and to quickly
involve operators when necessary. The desecration falls into two categories: agents must be able to run within the network without being a drain on it and agents must be able to communicate with the principal without being burdensome.

Each of the example systems has a natural time frame in which it needs to run. Alice’s shopping list manager can run on a daily basis. The aware alarm clock needs to run a few times an hour. The network monitoring system needs to run on a sub-second timeframe to identify unusual activity and start enacting a response.

In the first two examples the load on most principal’s computers is minimal, a few seconds at a time and in both cases can run when the processor load is low. The Proactive System Operator may cause problems if it is run on a principal’s workstation where it may take up significant processor time, possibly enough to cause the principal to feel that their other software is running slowly. If the system is running on a general server it may affect the server’s services. Even if the system is running on a dedicated server its queries to other services may put a load on the network which is noticeable. As such, the system needs to be implemented and deployed in such a way that it is still able to quickly identify and report problems without being a problem itself.

Similarly a personal agent system should not put too much cognitive load on to the principal. Suppose the network was being probed by an outside source. This would result in a number of alerts within a short time frame as the agents notice the unusual activity. If each of those alerts are presented to the principal one at a time, then the principal is bombarded with information. This distracts the principal, and it leaves them with more information to sift through to discover the cause of the alerts.

The personal agent system should be able to involve the principal in a manner proportional to the situation. If the network is being probed, then the principal may need a small alert in their messages or even an e-mail. If the system is breached, then the principal may
need a prominent alert. In order to understand the situation well enough to give appropria
tie responses, there must be agents in the system that are able to process and interpret
information from the monitoring agents.

The definitions for the Task and Workslip Scheme provide a broad concept of what
personal agent systems can do for a principal and how they can be constructed. The philo-
sophical guidelines provide restrictions on that concept to limit the systems created to ones
that will benefit the principal. In the following chapter TWPARI-J is introduced as a ref-
ence implementation of the scheme, which will show how both the definitions and the
philosophical guidelines work and interact.
Chapter 6

TWPARI-J: A Reference Implementation for Personal Agent Systems using the Task and Workslip Scheme

The previous chapters explored the core concepts necessary to create a personal agent system using the Task and Workslip Scheme and the philosophical guidelines that should steer the creation of those systems. This chapter explores how those concepts and guidelines can be implemented in the real world to create real systems. The Task and Workslip Personal Agent Reference Implementation in Java (TWPARI-J) is an example implementation, which demonstrates the scheme. TWPARI-J provides a framework, and the tools necessary to construct agents, which can be used to construct personal agent systems.

There are three main concepts implemented in TWPARI-J. These are the workslip, the agents, and the Infrastructure that connects them. The workslip provides the data structure
that agents are able to build, get data from and share. The agents provide the tools to receive workslips, process them however the developer needs, and then send them on to other agents. Finally, the Infrastructure provides the tools necessary to help agents communicate and to help developers do tasks that are commonly needed when constructing personal agent systems.

TWPARI-J is not the only way that the Task and Workslip Scheme can be implemented. It does, however, show how the concepts and guidelines can be implemented in such a way that a developer can easily construct agents, and a principal can easily construct personal agent systems to use them in their own way. It also allows for a concrete discussion about how the concepts and guidelines face the issues of the real world. This discussion is extended in the following chapter where three systems constructed with TWPARI-J are presented.

TWPARI-J is implemented in Java using the Java 1.8 SE specification. It was implemented in Java to take advantage of Java’s portability, maturity, support and community. It also makes use of the JDOM 2[Thea] library for managing XML files, and the JodaTime[Theb] library for managing time. These commonly used libraries were selected to improve the ease of use and the portability of the framework in creating agents.

Thus far, personal agent systems have been described as a set of agents which are somehow able to communicate with each other and which are able to work on a particular task, after receiving a workslip. In TWPARI-J, personal agent systems largely follow this structure. Each agent has some type of host that it is able to run on and it is able to send workslips to other agents as shown in Figure 6.1. TWPARI-J introduces a dispatcher to simplify the process of communication. At setup time all agents send their master workslips to the dispatcher, which indicate which fields they will read and which they will update. At run time, all agents send their workslips to the dispatcher which sends the workslip on to all agents which want to read any fields which have been updated. The dispatcher does not
make any decisions, instead it only notes which workslip fields have changed, and passes those workslips on to interested agents. This allows for improved communication efficiency without altering the Task and Workslip Scheme.

There are many different ways to organize personal agent systems which could be used to create these systems. The dispatcher is only one, with its own benefits and drawbacks. As discussed, it was selected as the communication mode for TWPARI-J because it simplifies some aspects of communication and reduces the amount of work needed to construct new agents. Other arrangements may be preferable in other situations and generally in the rest of this discussion, when an agent sends a workslip to the dispatcher, it is only a little bit of extra effort for the agent to send the workslip on to the next appropriate agent(s) directly.

Agents are made up of three parts, shown in Figure 6.2, the agent itself, the agent core and the agent cell. The agent core is responsible for implementing the logic of the agent (and is the part of the agent that most developers will interact with). The agent cell is responsible for executing the agent, communicating with other agents, and accessing resources. The agent is responsible for managing the connection from the cell to the core.

The first part of this chapter discusses the workslip, how they are implemented in TWPARI-J and a format for sharing them between agents regardless of the implementation. It also briefly discusses the Master workslip and how it is used to help agents communicate about workslips. The second part of this chapter discusses the agent, how agents are implemented and how they are attached to resources and executed. The third section describes the infrastructure TWPARI-J provides including the dispatcher agent and the systems for communication between agents. The fourth section of the chapter describes how these components can be combined to provide a framework to an actual implementation of a personal agent system.

This chapter does not provide complete documentation for TWPARI-J, rather it discusses
Figure 6.1: A simple personal agent system. Here each agent connects to the dispatcher for communication and a potential flow of a workslip across the system is shown. In operation, as each agent finishes work it will send the updated workslip to the dispatcher which will determine which agents could work on the workslip next and it sends it on to those agents.

how and why the implementation was built the way it was, and how the implementation relates to the concepts and philosophical guidelines. The full framework documentation can be found in Appendix A.
6.1 Workslip

The workslip was primarily influenced by the idea of a job sheet attached to a product as it is being created on an assembly line in a factory. The implementation in TWPARI-J follows this idea by having the workslip include three parts: a generic “raw” element representing the work, a collection of fields describing information about the work done, and a collection of fields describing the metadata about the job, such as: a name, the time it was started,
when it was last worked on and which agents have worked on it. Agents pass the work and
the workslip along together in the same way something being built on an assembly-line is
passed from one work station to the next.

The raw element can be any piece of data that can be passed from agent to agent, or can
be a reference to a place where the “work” can be accessed. Consider a personal agent system
that monitors a scientific experiment using pictures taken, at a regular interval. When a
picture is taken an agent can process the picture for a “quick look” and determine if other
agents should be invoked. The agent can create the workslip fields it needs for its simple
evaluation and the raw data remains available for agents later in the system, so that they
can also look at the picture. This way there is also no obligation for an agent to collect data
for agents later in the system.

Alternatively, consider an artist who would like to post a new piece of art to several social
media services. Each of those services has a specific set of requirements that an agent can be
designed to manage. The Artist’s personal agent system can have agents configured for each
service they would like to use, as the system runs the workslip can be passed from agent to
agent and the art itself can be passed as the raw element that the workslip is attached to.

The metadata is used to refer to information about the workslip itself, rather than the
work the workslip describes. Principals and developers can add metadata as they need it,
but the four common fields included help agents identify workslips and ensure that they are
passed to the right agents at the right time. The name and the creation time are needed
to uniquely identify the workslip. The name is usually the name of the agent which created
the workslip and the creation time concatenated together. The modification time is updated
each time an agent updates the workslip and is used mostly for debugging purposes. As
agents finish working on a workslip they “sign off” in the agents field so that there is a list
of which agents have worked on a particular task instance.
Finally, the workslip has fields about the work being done. These fields hold information that agents will need to make decisions or to do whatever work they need to do. For example, the Aware Alarm Clock begins with the weather source agent reading in a weather report from Environment Canada. The agent then breaks down that report into necessary fields, such as current temperature, dew point, visibility, the forecast and other information such as when and where the update was produced.

Later, the weather evaluation agents update fields that include scores for the situations they watch for. For example the weather evaluation agent for snow updates a field which provides a score from 0-1000 based on how severe the snow will be. Later still, the alarm control agent updates fields that update the alarm time, based on how it interpreted the scores provided by all of the evaluation agents.

Each workslip field comes in two parts, following the workslip definition (see Section 4.3), the field information and the field value. The field information includes a name, a type (usually a Java Class), a potential value (described as a string which a checker can later use to check the value) and a human readable description. The value then contains the information that agents have stored in that field (if any information has been stored yet).

Within TWPARI-J the workslip takes two forms. It has a representation within agents where it is used as a Java data structure that agents read and update, and it has a XML format which is used to share workslips between agents. There are also occasions when agents need to describe the fields they use in the abstract and in those cases there is a master workslip that includes the field information for all of the fields an agent might use, but no values. The following subsections discuss these two forms of a workslip, how it is implemented in Java, described in XML and then how the master workslip is implemented and used.
6.1. WORKSLIP

6.1.1 Workslip Implementation

At the most abstract level, the workslip in TWPARI-J is described by the \texttt{WorkslipInterface}. This Java interface describes the basic actions a developer needs to be able to do to use a workslip, access the raw work, read and write workslip fields, check if a workslip has a given field, list all fields and get the name for the workslip. A diagram of the interface is shown in Figure 6.3.

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{Workslip} \\
\texttt{<<Interface>>} \\
+ getRawValue():T \\
+ read(key):\texttt{WorkslipField} \\
+ write(key, data) \\
+ fields():\texttt{Set<WorkslipField>} \\
\hline
\end{tabular}
\end{center}

Figure 6.3: The \texttt{WorkslipInterface}, it requires access to the raw internal value, to read and write each workslip field and access to the workslip fields as a set.

\texttt{WorkslipInterface} provides all of the tools a developer needs to work with information for a personal agent system. Any field can be checked for existence (helpful to allow fields to be added if they have not been accessed yet or updated if they have been), any field can be written to, or read from, the workslip can be uniquely identified, the fields can be accessed en mass and the raw work can be accessed.

At this high level view, the workslip name provides the only metadata about the workslip and is mostly used to help identify workslips as they are passed out of the agent. All other fields in the workslip are treated as the same, whether they are metadata or information about the work. This description of the workslip is used particularly by the parts to the agent which only need to interact with the workslip as a store of data, not one that needs
any particular behaviours.

The class Workslip is the primary implementor of the WorkslipInterface and provides the bulk of the functionality that agents need to use the workslip. A diagram of the Workslip is shown in Figure 6.4. New workslips can be created with or without the raw element (if no element is provided, the reference is left null) and at creation time, the creation time is recorded, the last modification time is set to the creation time and the name of the workslip is created. The workslip name (as mentioned in the Previous Section) is simply the name of the agent creating the workslip concatenated with the creation time.

<table>
<thead>
<tr>
<th>Workslip</th>
</tr>
</thead>
<tbody>
<tr>
<td>- raw:T</td>
</tr>
<tr>
<td>- fields:Map&lt;WorkslipField, Object&gt;</td>
</tr>
<tr>
<td>+ getRawValue():T</td>
</tr>
<tr>
<td>+ read(key):WorkslipField</td>
</tr>
<tr>
<td>+ write(key, data)</td>
</tr>
<tr>
<td>+ contains(key):boolean</td>
</tr>
<tr>
<td>+ fields():Set&lt;WorkslipField&gt;</td>
</tr>
<tr>
<td>+ checkForErrors():boolean</td>
</tr>
<tr>
<td>+ handleDefinedKeys(key, data)</td>
</tr>
<tr>
<td>+ getCreationTime():DateTime</td>
</tr>
<tr>
<td>+ getModifiedTime():DateTime</td>
</tr>
<tr>
<td>+ getWorkslipName():String</td>
</tr>
</tbody>
</table>

Figure 6.4: The Workslip, provides the primary methods needed to work with a workslip, including those defined in the WorkslipInterface, but also those for checking the correctness of the workslip, handling special fields and managing metadata types of information, such as the creation and modification times and and the workslip name.

The collection for workslip fields is also initialized as the workslip is created. This is implemented as a Map, which pairs the workslip field information with the value that field has, which allows for any field to be accessed at any time and for the value to be null if it
has not been defined by the agent.

The structure can be accessed through the Workslip, when a field is read or checked to see if the workslip contains it. These queries are passed on to the map to return the value or the field’s existence. When updating a field, the process is similar, but the Workslip object first checks to see if there is a problem with the value, running a method called checkForErrors and then runs a method handleDefinedKeys to check if the field has been designated as special, before updating the value in the workslip field map. When the workslip is updated, the modification time of the workslip is also changed to current time.

Special fields usually have explicit accessors and mutator methods which make accessing the field simpler in code. Most special fields are the metadata and so name, creation time and modification time are all held this way. Other than this designation, there is no internal difference between metadata fields and normal fields. The Workslip class also uses an internal prefix to help identify metadata fields.

The method checkForErrors is used primarily to check that a workslip’s value matches its given type and potential value. It can also be used if there is a more technical problem that would result from setting a workslip to that particular value.

**Workslip Fields**

In Section 4.3, the workslip \(w(\text{Ins}(Ta,t))\) is defined as being made of two parts: the information which describes the field, and the value which contains the current value of the field. The WorkslipField is used to hold the workslip field information of this structure. It is combined with a value through the workslip map to form an actual workslip that agents can use. A UML diagram of the class is shown in Figure 6.5.

Internally the WorkslipField contains the field name, the potential value, and the field description as a string and the field type, as a Java type. The Workslip file is expected to be
6.1. WORKSLIP

<table>
<thead>
<tr>
<th>WorkslipField</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: String</td>
</tr>
<tr>
<td>- type: Class</td>
</tr>
<tr>
<td>- potentialValue: String</td>
</tr>
<tr>
<td>- description: String</td>
</tr>
<tr>
<td>+ getName(): String</td>
</tr>
<tr>
<td>+ getType(): Class</td>
</tr>
<tr>
<td>+ getPotentialValue(): String</td>
</tr>
<tr>
<td>+ getDescription(): String</td>
</tr>
</tbody>
</table>

Figure 6.5: The WorkslipField, houses the information needed for a field in the workslip including a unique name, a type, a restriction on potential values and a human readable description.

created once, and therefore is immutable after it is created. Constructors allow the values to be set individually or to be parsed from a information string, but after they are set, the values can only be accessed, not changed.

The type and potential values of a field can be enforced through the checkForErrors method in the Workslip class. A developer can create/use a general method to ensure that the workslip is in a form they expect, by extending the default checkForErrors. This approach does allow developers maximum flexibility, and the power to enforce any rules about data needed for their agents.

6.1.2 Shareable Workslip Format

The Workslip class works well within agents in TWPARI-J to represent the workslip, but to maximize compatibility between agents, regardless of implementation, agents always share workslips using a common workslip format. The common format uses an XML document to describe all of the fields in the workslip. Libraries for reading and writing XML documents
are common, which makes it an ideal way for agents to write, read and transmit workslips.

The XML document is as simple as possible for each workslip. Each document contains a Workslip element, which contains a metaData element and a workslipEntries element. The meta-data element contains entries from the workslip name, creation time and modification time. The workslip entries element contains an element for each field of the workslip. These workslipField elements contain two: the workslip field information including the name, type, potential value and description, and the workslip value containing a representation of the value. The full documentation for the XML structure can be found in Appendix C.

This system works well for simple data types, but if developers want to use more complex structures or objects, then there needs to be a common way to serialize those data types. For the three example applications using String to contain “flattened” information has largely proved sufficient.

The Shareable Workslip Format is sufficiently general to ensure that agents can read and update fields regardless of how they are implemented. It is also portable across operating systems and can be transmitted using a wide variety of methods.

6.1.3 Workslip Lifecycle

Workslips are generally not created from scratch. Instead they usually begin life as an empty workslip and then agents updates fields to them as necessary. In the case of an agent that does not read other fields to begin working (for example, the Weather Source Agent) they simply update the fields they update as they produce them. In the case of an agent that receives fields from another agent, the agent will use a WorkslipSerializer which will take an empty workslip object and updates the field information and values from the Shareable workslip format. Once the workslip has been ‘inflated’ the agent begins working and updating
its own fields.

When the agent is finished working on a workslip, it uses a `WorkslipSerializer` to convert it into the Shareable Workslip format and then sends it to the dispatcher.

### 6.1.4 Master Workslip Implementation

As with the core concept of the Master Workslip, each agent needs to be able to discuss the fields it works on in the abstract. As an agent begins working, one of the things it has to do is compile a list of all of the fields it will access over the course of its execution. This includes fields it will update regularly and ones that might only be updated on rare occasions.

This collection is implemented within each agent as a list of `WorkslipField`s. This list is then used to streamline, and codify agent access to workslip fields. When an agent accesses a field, it first retrieves the `WorkslipField` object from the master workslip through a method which looks up the fields given the field name. This ensures that the correct field is being updated by the agent. This also allows for developers to define their own name replacement system to ensure that fields with different names in different agents can still be used in the same way if they represent the same information.

The master workslip is used for a few other purposes, particularly relating to agent identification. When agents start working together they share their master workslips so that other agents (or the dispatcher) will know which fields they are interested in. It is also used to help developers identify and debug agent behaviour.

### 6.2 Agents

As it is formally described (see Section 4.6) an agent $\Lambda_g$ really represents a function, $\text{action}_{\Lambda_g}$, which describes an update to a number of workslip fields, (the output) based
on the values of a number of input fields (the input) so long as some conditions are met in the world. At the highest level of abstraction within TWPARI-J this is reflected in the core interface `Agent`, which requires that an agent `takeInput` in the form of a `Workslip`, work to do whatever it does, and then `produceOutput`, providing a `Workslip`.

The `Agent` interface provides the basic actions that an agent will do, but does not cover all of the theoretical and real world responsibilities an agent will have. To describe the conditions under which it should work, each agent must be able to provide `AgentInformation` which identifies the agent, but also lists the fields it will need and the conditions under which it will work. Each condition is described as a particular set of values that a field can have. This `AgentInformation` is used then either by the agent’s cell (which allows agents to execute and links them with resources) or the special dispatcher agent to determine if any particular workslip is worth giving to the agent to work with.

The `AgentCell` provides the entry point for each agent and links them with necessary resources. These may include simple resources such as the agent’s configuration or more complex resources such as sensor information. In order to handle the different ways in which a developer may want an agent to run, the `AgentCell` can use several different `AgentExecutionHandler` to modify the way in which the agent is executed.

When it is executed, each agent must first setup its `MasterWorkslip`. After that point, the agent queries the master workslip to get the correct field information for that field (this ensures that each field is unique within the agent and that developers cannot accidentally create or access a new field mid-processing). After that, each agent must be able to take a workslip as input, work on that workslip, then update that workslip and provide it as output.

Specifics such as how a workslip is transmitted to an agent or sent on or how often an agent is invoked to work are handled by the `AgentExecutionHandler`. The logic of how an agent sends and receives workslips is usually isolated from the processes of the actual “core”
logic of the agent (the agent then compares shopping lists to sale information still handles
workslips similarly to the agent that identifies if an ftp server is not running). To manage
this, this core logic is usually abstracted as the AgentCore.

In order to look more deeply at the tools TWPARI-J provides for developers, it is good
to begin by looking at how configuration information is passed to the agents. Then it will
be possible to explore the AgentCell in greater depth, before looking at agents themselves
and how a developer will create their own new agents.

6.2.1 Agent Configuration

Agent configuration is necessary to allow agents to find the right resources and do the right
things at runtime. When running an agent queries its configuration to determine how the
principal would like it to operate. Basic configurations include things that all agents will
need such as the location of the configuration file and the class of the agent that is going
to run. Configuration also includes specifics for each agent such as which URL it should
retrieve the weather report from, where the grocery list is or the e-mail for the operator.

Configurations are created in layers. The first is the default layer, provided by the devel-
oper when designing the agent. They can provide default values for any configuration which
the agent will use later, unless overwritten. They can also omit any optional configurations,
or configuration the principal must provide later (such as login information). These are en-
forced by the behaviour of the agent, if it cannot operate without a configuration then it fails
with a message explaining the missing configuration. If the configuration is optional, then
there may be a low priority log message or the agent might simply treat that as a normal
case.

The second layer is the configuration provided by the principal when designing their
6.2. AGENTS

personal agent systems. As they set up each agent, a principal can provide further configurations in configuration files to fill in any required configurations and to change any defaults or optional configurations. At runtime the agent overwrites any default configurations with configurations from the principal’s file.

The third layer is the configuration provided by the principal in the arguments to the agent when it is started. These configurations are treated as highest priority and override any configurations in the file or any default configuration. This allows the principal to adjust on a run-by-run basis even when they have a usual configuration they would like to use.

Configurations are stored as name, value pairs in the ConfigurationSet class. Each value is kept as a string, and if a developer needs to use the configuration as a specific type they must manually convert to the appropriate type before using the value. The agent cell makes the configuration available to the agent and to retrieve a configuration value it must query the ConfigurationSet with the name of the configuration it would like. If the configuration exists, the value is returned, if it does not, a MissingConfigurationException is thrown instead. It is the developer’s responsibility to handle the missing configuration in an appropriate way.

At the beginning of execution, the agent cell begins running, and loads the default configurations the developer has provided. It then takes all of the configurations it was passed on the command line. Each command line configuration is in the format \texttt{name=value}. Then the agent checks for the configuration for the configuration file (which can be the provided default, or overridden by the command line configurations) and uses that configuration to attempt to load the configuration file. If the file exists and the agent can read it it then fills the ConfigurationSet. If a configuration has a default value, it is overwritten, but if the configuration was already set on the command line, then that value is used. This process is illustrated in Figure 6.6.
6.2. AGENTS

Figure 6.6: Agents can have three sources for configuration, default configurations that are built into the class by the developer, configuration file configurations which are setup by the principal when the agent is first set up and command-line configurations. Command-line configurations overwrite all other configurations, and the configuration file configurations overwrite the default configuration. The exception to this is the configuration for the path to the configuration file, the agent will always check the default command-line configuration and then the default configuration to try to find the path to the configuration file.

Once this is complete the agent cell and the agent should have all necessary information to begin executing.

6.2.2 Agent Cell

The agent cell is based on the work in [MGR+99], which proposed using a virtual machine to “house” agents. This allows agents to be written more simply, but still take advantage of the particulars of any machine that it is running on. Hive is discussed in more detail in Section 3.4.3.

The agent cell provides the ability to execute an agent. It provides the entry point for the JVM to execute, and then it uses reflection to bring in the necessary components to run the agent. In order to be as flexible as possible, the cell uses one of several different AgentExecutionHandlers which contain different implementations for executing the agent. These include handlers for normal execution, for testing, for allowing the agent to register with the dispatcher, or for simply providing diagnostic information.
6.2. AGENTS

A generic AgentCell begins executing by taking the configurations off the command line and adding them to the ConfigurationSet. It then looks for the configuration file, and, if possible, loads the configurations from there as well. Once the custom configurations and any other necessary resources have been loaded, the AgentCell looks up the correct AgentExecutionHandler and Agent. These are then loaded dynamically through the Java Reflection API. If both are successfully loaded, the AgentExecutionHandler is pointed at the Agent and allowed to run. When the execution model is finished running the agent, the AgentCell cleans up any open resources and terminates.

The AgentCell provides resources to the Agent through the static class AgentResources. This static class is created with the AgentCell and provides access to several different resources including the ConfigurationSet, the MasterWorkslip, the AgentInformation and the WorkslipSerializer. It also includes other commonly needed pieces of information such as the names of metadata fields and commonly needed configuration names. This simplifies agent design by providing a common place for common information that is always accessible.

Agent Execution Handler

While the cell provides access to resources, the actual execution of the agent is deferred to the execution handler. Each of the different execution handlers provides a different way to execute the agent in a way that is necessary or helpful in a personal agent system. Each Execution Handler extends the AgentExecutionHandler interface which requires that they each provide a method run which will execute the Agent.

For the set of personal agent systems built to demonstrate TWPARI-J several execution handlers have been needed. These include:

- the EmptyExecutionHandler, which outputs a single empty workslip.\(^1\)

---

\(^1\)Agents which have no fields they need to read must still be prompted to begin working, this is done with
6.2. AGENTS

- the **VersionExecutionHandler**, which outputs the **AgentInformation** which includes the agent name, the issuer, the author, the version number, a contact method, an agent description and the MasterWorkslip for the agent (augmented with which fields the agent will read, and which it will update). This is usually used for debugging or logging purposes and can be used to produce a short, long, or very long version. The short version includes only the basic agent information, the long version includes the master workslip and the very long version also includes the agent configuration.

- the **RegistrationExecutionHandler**, which identifies the agent to the dispatcher. This identification includes the **AgentInformation**. The exact method of communicating with the dispatcher is deferred further to a **RegistrationMode** which is responsible for transmitting the information. After the registration information has been sent, the agent shuts down.

- the **SingleExecutionHandler**, which runs an agent once processing a single workslip. This is primarily used for testing, but can also be used for situations where it is not desirable for an agent to sit waiting to act.

- the **ContinuousExecutionHandler**, runs an agent continuously processing workslips. This is the primary method in which agents are run in a personal agent system that is working normally for a principal.

The agent execution handlers provide a common way to run an agent and to do each of the additional tasks that an agent may need to do. This allows the **AgentCell** to be focused on gathering resources to pass to the agent and to execute the agent in the same way regardless of which agent and which cell are needed for the personal agent system.
The cell and the execution handler further allow the agent to be designed in such a way that it is isolated from the necessities of execution. An agent is only responsible for being able to act on a workslip, and it can access any additional information it may need through the resources provided by the cell.

### 6.2.3 Agent Implementation

Each agent needs to be built around the concept of transforming an input workslip into an output workslip, based on the information in that input workslip and on the state of the world as the agent perceives it. With these details of execution omitted, an agent should be fairly simple to implement, provided, especially, that it solves a well-suited task step for a personal agent system.

Each agent must implement the `Agent` interface and provide methods to take a workslip through `takeInput`, work on it through `work` and produce the next workslip through `provideOutput`. Additionally each agent needs to be able to provide its default `ConfigurationSet` and its `AgentInformation`.

While the `AgentCell` omits the details of running the agent, it is often possible to separate the remaining responsibilities of the agent into two groups, the administrative responsibilities of receiving and producing workslips and the production responsibilities of doing the actual work of the agent. For this reason many agents defer the actual work of the agent to an `AgentCore` which encapsulates the actual logical work of the agent free of the context of the workslip. This is useful for those cases where that encapsulation makes writing an agent simpler but can be omitted if it is an unnecessary step.

In execution, an agent needs to be able to begin by setting up its default `ConfigurationSet`, its `MasterWorkslip` and its `AgentInformation`. After the agent object is con-
After this, at some point in time, the execution handler will provide the agent with a workslip through the `takeInput` method. This method needs to unpack the necessary information from the workslip and ensure that it is in the correct state for the agent to work on. Usually this involves taking the values stored in particular fields in the workslip and storing them in local variables (of the correct type) in the agent itself.

Next, again at some later point, the execution handler will call the agent’s `work` method. At this point whatever the agent’s internal logic is is executed. At this point it is not important to consider these details as there will be many examples of agent work in the following chapter looking at the personal agent systems built using TWPARI-J.

Finally, again later, the execution handler will call the `ProduceOutput` method. Then the agent will take the data produced while it worked and package that internal information into the workslip so it can given to other agents.

This approach is intended to simplify the creation of personal agents as much as is possible. In this way a developer only needs to find the necessary information in the correct set of fields, process it with resources provided by the `AgentCell` and then put the result into a workslip for the next agent to work on. It enforces encapsulation and encourages each agent to be as simple as possible.

### 6.3 Infrastructure

The `AgentCell` provides the bulk of the infrastructure that the personal agent actually needs to run, but there are two further factors that the agents need in order to form a personal agent system. The first of these are systems to handle communication between agents and
6.3. INFRASTRUCTURE

the second is a routing system to transport workslips between agents in the system. This section outlines both of these concepts as they have been implemented in TWPARI-J.

As mentioned, TWPARI-J relies on the dispatcher to act as a workslip routing agent, which is responsible for ensuring that each agent receives workslips they are interested in working on. TWPARI-J uses two particular mechanisms to allow agents to communicate with the dispatcher. These mechanisms could be also used to allow agents to communicate directly, with a little reorganization on the level of the AgentCell. That being said, TWPARI-J is built on a dispatcher centric model, which essentially decentralizes each agent so that its selection of work algorithm is running on the dispatcher agent, while the rest of the agent is running on its own. This allows for significantly reduced communication overhead (no sending agents workslips they will not work on) and it provides a central place to handle conflicts which may arise from several agents working on a workslip at the same time.

6.3.1 Communication

There are a great number of technologies that exist to make it simple for two pieces of software to communicate. The modular nature of the AgentCell makes it easier to integrate any of these technologies since a cell only needs to receive workslips which can be passed to an agent and then take the production of the agent and send it on. TWPARI-J provides one mechanism for agent communication, which relies on file passing. In future versions of TWPARI-J it will be simple to implement different communication protocols for agents to use, via their cells.

This method of communication arose primarily due to the test environment for TWPARI-J and the implemented systems. In these cases, agents were frequently being executed on
the same host, or on multiple hosts which shared a network file structure.

This allows the existing infrastructure to handle the network communication (if necessary), and provides a simplified way for agents to communicate. Agents simply need to be assigned an input directory in which the dispatcher (or other agents) can write workslip files. They can then check those directories for workslip files, read and work on them and then put their output in the directory for the dispatcher. As an added convenience, this provides a simple system for buffering as well.

The specifics of file passing communication are implemented in an abstract class \texttt{FilePassingAgentExecutionHandler} which implements the \texttt{AgentExecutionHandler} interface. This abstract class provides utilities to read and write workslips as files, as well as utilities to manage collecting workslips from the filesystem and organizing them for the agent to process. The \texttt{FilePassingAgentExecutionHandler} is then implemented by classes which provide either single or continuous running to the agents they are executing.

The \texttt{FilePassingAgentExecutionHandler} model is to collect file handles for all of the workslip files in the agent’s given input directory. Then each file is read, inflated, using the \texttt{WorkslipSerializer} to produce usable workslips for the agent and then deleted from the file system. The agent is then given the new workslip, allowed to process it, and then produces an output workslip. The output is then flattened into an XML file using the \texttt{WorkslipSerializer} and written out to the agents given output directory (which is usually the input directory for the dispatcher). When there are many input workslip files the agent loops over all of them before looking for further input files.
6.3. Dispatcher

The dispatcher is a specialized agent that moves workslips from one agent to another. The dispatcher receives registrations from agents as they begin running and then uses that registration information to help route workslips from one agent to another. The dispatcher is intended to reduce the amount of code that an agent developer might need to write and to simplify the process of creating a personal agent system for the principal.

The dispatcher shares many features with agents and agent cells, including the same system for configuration and the use of handlers to specify its particular operation, especially the processes for receiving registrations, receiving workslips and sending workslips. The specifics of routing workslips between agents is handled by the `WorkslipDispatcher`.

The dispatcher begins operating much like an `AgentCell`, by loading its configuration. As with the `AgentCell` it first consults its command line configuration and then loads its configuration from file where those values do not conflict with the command line configuration. After that the dispatcher loads three handlers through reflection. The first handler, the `RegistrationHandler` is responsible for receiving registrations. Specific implementations either start a server to receive network connections or look for registration files in the registration directory.

When agents register with the dispatcher (through whichever method) they provide agent information which identifies the agent and provides coordinates where the dispatcher can send workslips that agent would be “interested” in. Agents also provide their master workslips, annotated with which workslip fields that they read or write. The dispatcher uses this to determine when an agent should be passed a workslip.

The second and third handlers are the `WorkslipIntakeHandler` and `WorkslipOutputHandler`. These are respectively responsible for receiving workslips from agents and then sending work-
slips on to following agents. As with the RegistrationHandler specific implementations either use the network to communicate or the file passing system.

When the dispatcher receives a workslip from an agent it consults updated fields in the workslip and compares these against its internal list of agent interest (that have not already received this workslip and then sends a copy of the workslip to each agent in that list\(^2\).

While the dispatcher may seem to be offering a central control, it really only offers a service to the agents. Each agent would otherwise have to find all other interested agents to send a workslip to. Because the dispatcher only handles information for agents registered with it, it is effectively distributing that part of an agent’s operation.

### 6.3.3 Scripts

The final part of the infrastructure provided by TWPARI-J is a set of scripts to simplify the execution of agents. These include a few simple utility scripts that move workslips around, but also include scripts to simplify agent execution. They provide a common interface to access agent configurations and to point agents at their configuration files. They also provide convenient shortcuts to run agents in each of the different execution modes (such as to produce an empty workslip, report version, register with the dispatcher, and run either for one workslip or continuously).

The scripts are designed to run in a Linux environment using the \texttt{bash} shell. As such while they represent the basic execution support a personal agent system will need, they cannot be considered a complete utility for all principals and developers.

The first of these scripts is \texttt{run\_agent.sh} which is responsible for executing the cell class

\(^2\)This excludes agents which do not need to see the workslip again. The agents update a special field with information when they are done, so that the dispatcher will not return the workslip to them if it is not needed.
and providing the necessary configurations (usually by providing the path to the configuration file). This script is called by a number of different scripts that execute the agent in its different execution modes: empty.sh, which produces empty workslips, register.sh, which registers with the dispatcher, run.sh which executes the agent ready to receive and process workslips continuously, test.sh which executes the agent once, and version.sh which causes the agent to print out information about itself.

Additionally, there are scripts to execute a full personal agent system. One example is setup_alarm_clock.sh which registers and then runs all of the agents needed for the alarm clock system. Once the system is set up the push_workslip.sh can be used to push an empty workslip to the first agent in the system (the WeatherSourceAgent) to allow it to begin.

6.4 Developing Agents with TWPARI-J

The following chapter contains examples of agents created using TWPARI-J as a base framework and of systems put together with those agents. Before looking at those agents though it may be useful to look at the steps necessary to create an agent. The WeatherEvaluation-Agent from the Alarm Clock system requires information in a workslip regarding the weather conditions and produces information in a workslip regarding its evaluation of the weather conditions.

As with most software, designing an agent begins with gathering requirements. From there it is necessary to check the workslip fields the agent will need to read and which fields it will update. With the agent’s role within the system established (with respect to which fields it will read and update), the actual operation of the agent can be designed.
6.4. DEVELOPING AGENTS WITH TWPARI-J

6.4.1 Planning the Agent

As frequently mentioned, agents are effectively a function to transform information. This means that one of the best starting points when developing an agent is to check the information that needs to be transformed. For the \textit{WeatherEvaluationAgent} the agent is going to evaluate the weather report provided by another agent and will produce one or more evaluations of the report\textsuperscript{3}.

In order to do this, the agent needs to be able to extract important information from the report and then compare it against some criteria to determine if the weather is “bad” or how bad the weather is. In order to do this the principal needs to be able to provide some rules to the agent in order to tell it what information to look for and how to determine if that information is bad (or how bad). This can be managed by the developer providing some specialized logic that the principal can configure.

In particular it would be nice to have a system to handle keyword data and a system to handle scalar data. For example if the weather report had a forecast including the word frost in the next 24 hours that should be reported as bad. As another example if 2cm of snow have fallen in the last 12 hours that should get one score of “badness” whereas if 50cm of snow have fallen that should get a different score.

To keep this information simplified, the evaluation can be rated on a 1000 point scale, which can be used as the value for the field that the agent updates with its evaluation. If the agent is evaluating for the existence of a keyword then it can provide a static score depending on the principal’s wishes (frost may receive a 500, whereas blizzard may receive a 1000). For scalar data the agent can map its internal severity measure onto the 0-1000 point range.

Using this as the general plan for the agent, there are two factors that need to be decided

\textsuperscript{3}Ideally to follow the scheme, each evaluation agent should be responsible for a single evaluation
before work on the agent can begin. The first is to decide exactly what fields the agent will use and the second is to provide an algorithm for the agent to follow.

The Workslip Fields

It is important to remember that not all principals and not all developers will name fields the same way. As such, this means that all agents must contain a translation mechanism so that if the principal feels one field is equivalent to another, they can provide a translation equivalence so the agent will be able to look for fields by other names. That being said, here it is sufficient to assume that all agents will know the same fields by the same names.

On the reading side, the agent needs to find fields containing the weather report. In particular the agent looks for several fields, including:

- `weather.current.temperature` - which holds the current temperature (as a number),
- `weather.current.condition` - which holds the current weather conditions (as a string),
- `weather.warnings` - which holds any warnings the weather service may be reporting (as a string), and
- `weather.forecast` - which holds the next several days forecast (as a series of strings).

The agent also needs to update fields with the values it evaluates. This varies with what the agent is actually doing, which is defined by the evaluators (explain further in the next section). In general the agent updates either the field `weather.eval.<evaluation_name>.keyword.score` or the field `weather.eval.<evaluation_name>.value.score`, both of which hold a score from 0 - 1000. The evaluation name is given in the agent configuration to allow different evaluations to be uniquely identified.
The Agent Operation

With the updates to the workslip designed, there is little that remains for the developer to design within the agent. The agent needs to be able to handle workslips, obviously, which should be fairly standard for the developer to model from other agents. Further the agent needs to be able to load the correct components to handle the evaluation that it is intended to perform. There are two evaluations that seem necessary, the first is a numeric evaluator which can search for particular numbers and the second is a keyword evaluator which can search for a particular word in text.

The numeric evaluator will produce the value_score results and will need to be able to determine if a number (such as a temperature, or a precipitation amount) is greater than or less than a particular amount and by how much. This must then be translated into a rating on the 0-1000 value score which can be updated in the workslip. The developer may plan a rough translation but the exact values should be left to the principal when they are configuring their system.

The keyword evaluator will produce the keyword_score result and will need to be able to determine if a particular word (or set of words) is present in a block of text (such as a condition statement or a forecast). Instead of a translation the result is binary, and so the evaluator just needs to provide a score set by the principal for a particular keyword (for example, snow results in a score of 400, frost of a score of 300 and blizzard a score of 700).

The final piece of design that the developer will need to do is to determine how the evaluators will be loaded into the agent to work. One design choice would be to develop two different versions of the agent which perform each kind of work. Another design choice would be to have one agent which dynamically loads different code for each component. The first helps a system hold more strongly to the guideline of keeping agents as simple as
possible, but the second makes it simpler to reuse an agent with a different type of evaluation
and which helps make the agent more personal in the event that the principal would like a
refinement of the way weather evaluation is done. For this reason, the second way seems
preferable and is what the developer chose, but there is no reason why the first way could
not be used either.

6.4.2 Developing the Agent

TWPARI-J provides many resources to reduce the work a developer needs to do to create a
new agent. The default AgentCell allows an agent to run and communicate in the majority
of environments with no modification. Agent provides the infrastructure needed to read, and
update a workslip, and again for the majority of situations, no modification will be necessary.
This leaves AgentCore as the primary element of the agent where the developer will need to
add logic, taking the data read in from workslips, processing it and then updating workslips.

The Agent Core

In this case, the developer has decided to implement the core with dynamically loadable
evaluators. When being loaded, the core must dynamically load the evaluators based on the
agent’s configuration. When the core is invoked with a workslip, it will pass the workslip to
the evaluator, and update the workslip with the result.

There are several evaluators implemented, which are used in Section 7.1 as part of the
Aware Alarm Clock. The two simplest evaluators, the KeywordEvaluator and the ValueRangeEvaluator are explored here:

- the KeywordEvaluator loads a set of key words from its configuration. When it is
  invoked it checks if any of those words are contained in the field it is configured to
6.4. DEVELOPING AGENTS WITH TWPARI-J

read.

- the ValueRangeEvaluator loads a mapping scheme from its configuration. When it is invoked, produces a score based on the value in the field it is configured to read.

The body of the agent consists of the AgentCell and the evaluators. To implement them, the developer only needed to write a limited amount of code. The flexibility the provide makes it simple for a principal to configure the agent to evaluate the weather according to their own criteria, and add it to their personal agent system.

6.4.3 Running The Agent

To run the agent, the principal first needs to set the configuration for it. This includes basic information like the agent class and particular information such as the evaluator to be loaded, the settings for that evaluator and the field they would like to be evaluated. Once that information is ready, the principal can add the agent to their system and as long as they have an agent providing weather information, the WeatherEvaluationAgent can provide evaluations for a control agent which can then update an alarm agent.

In fact, the principal will probably want to have several instances of the WeatherEvaluationAgent running each evaluating a different aspect of the weather and having the control agent amalgamate that information in its decision making.

Ideally the system for a principal may look something like this. With an agent providing weather information, the principal can have several evaluation agents:

- one looking at weather.current.temperature to see if the temperature drops below \(-5^\circ\) and increasing the score by 100 for every \(5^\circ\) colder,
6.5 PERSONAL AGENT SYSTEMS

- one looking at `weather.current.condition` to see if the keyword “snow” appears in it which would give a score of 300, or 600 if the keyword “blowing” also appears,

- one looking at `weather.warnings` again looking for the snow keyword but this time giving a score of 1000, since snow in the warning is a particularly bad sign, and

- one looking at `weather.forecast` for words like snow, frost or rain with a low evaluation maybe 50 for situation where the weather is not as extreme as the above agent instances are looking for but where the principal may still want the alarm altered slightly.

With these in place, the principal can then adjust the control agent to consult the scores coming from all of the evaluation agents and adjust the alarm clock accordingly. The principal may allow 15 minutes for a score between 0 and 200, an hour for scores greater than 500, with gradient steps in between.

6.5 Personal Agent Systems

Planning and developing the `WeatherEvaluationAgent` works, provides a basic concept of how a Personal Agent System will need to be constructed. For each task the principal would like a personal agent system to undertake, they must complete several steps.

First, they must select the agents that will work on each of the task steps. For each task step there may be one agent, or there may be several that either act as redundancies, collaborators or alternatives.

---

4Note there is no particular relevance to the scale of the scores, they are configured by the principal and as only need to be consistent between agents in the principals personal agent system.
Then each agent must be configured to work on the task in the way the principal would like it worked on (and also to work with the other agents). This will include general requirements such as which agent classes will be loaded, which other components those classes will use and how (and where) the agents will communicate. This will also include specifics regarding how each agent works and the problem the agent is working on, such as the weather source location, the evaluation settings, or the alarm sound.

Once the configurations are set, the principal can set the agents to run using the scripting system and provide a dispatcher for the agents to communicate through. Then as agents begin running, they will first register with the dispatcher, sharing their master workslips and which fields they will read and write and then they will wait to receive workslips.

The system will begin running by passing an empty workslip to an agent. In the case of the alarm clock, the workslip is empty as the source agent only needs a prompt to go and check the weather sources. The same is true of the flyer agents in the shopping list system. Other agents want specific information such as the shopping list agent, which will receive input from the principal and use that to generate its starting workslip.

Once agents begin running they will update the workslips they receive and pass them on. The dispatcher receives workslips from agents and then compares the fields updated to its registry of agents. If there are several agents interested in receiving the workslip the dispatcher will send the workslip to all of them.

The personal agent system should be able to run continuously once it is started, for as long as the principal would like it to run (assuming of course that the hardware and outside worlds remain functional and cooperative). The only changes that should need to be made to the system are changes the principal wants to make in how the task is solved.

This chapter introduced the TWPARI-J framework, to help explore the three concrete example personal agent systems discussed in the following chapter. Each of the example
system demonstrates different uses of the workslip and infrastructure and different implementations of agents.
Chapter 7

Three Example Systems Using TWPARI-J

With the concepts of the Task and Workslip Scheme, the Philosophical Guidelines and the development framework provided by TWPARI-J explored, it is now possible to look at some applications of personal agent systems. In this chapter there are three systems discussed.

These systems are real-world solutions that match the tasks first discussed in Chapter 2. Namely,

- an alarm clock that is able to check weather and traffic reports and update the time it sounds,
- a shopping list manager that is able to check electronic flyers from stores and compare it to the principal’s shopping list to produce an efficient list for getting groceries,
- and a system operator that is able to monitor several systems to ensure that they are working properly and is able to alert its principal or to manage problems itself if it seems like a response from the human will take too long.
7.1. AN AWARE ALARM CLOCK

Each system exploration begins with a discussion of the task itself and then the requirements of the system. It then continues with a breakdown of the information the workslip for the system will need to contain. That information is then used to identify which agents will be needed for the system and how those agents will need to function. Finally, that information is used to look at how the system is organized and applied to do the task, as well as some discussion of how the task relates to the scheme.

Each of these aspects of the system design is broken out, to show how these systems were designed and constructed, and to provide a model for building systems using the Task and Workslip Scheme. Each system will be presented to show how it demonstrates the Task and Workslip Scheme in action.

7.1 An Aware Alarm Clock

As outlined in Section 2.1, In The Aware Alarm Clock a principal would like to adjust when their alarm clock sounds in relation to ongoing real time events (namely weather). For example, a principal might want to adjust their wake-up time earlier by 30 minutes if there was fresh snow the night before. In general, the principal will want to check one or more sources for weather information, evaluate the information provided and decide if they want to change their alarm time. The principal will want to assign this task to a personal agent system which will automatically perform the task as it is needed.

This task can be broken down into four subtasks. The first subtask is to retrieve the environment information from a source. The second subtask is to evaluate the environment and produce a digest of the environment information. The third subtask is to take the environment evaluation information and determine if it warrants a change of alarm time. Finally, the fourth subtask is to take the produced change of alarm time and adjust the
7.1. AN AWARE ALARM CLOCK

alarm for the principal (and actually sounding the alarm for the principal).

As a task, the Aware Alarm Clock is well suited to the Task and Workslip Scheme, and it illustrates the three attributes for tasks in the philosophical guidelines in Chapter 5. It is repeated, regular and informatic. The task is repeated, because this is a task that the principal will want running consistently overnight (repeated on a time scale between minutes and hours, and ideally at the same frequency as the weather reports are generated), and also the principal will want to run the task on most bad weather days where they will need to leave the house in the morning. The task is regular, because the output of the system is an update command to an alarm clock in the form of an adjusted time, the input is in the form of a weather report (which are consistently formatted) and internal information can be described as numeric scores provided by evaluating the weather report. Finally, the task is informatic, because all of the parts of the task are easily managed by a computer, from retrieving the weather report, to updating the alarm clock.

7.1.1 Requirements

As the broadest requirement of the system, a principal will require the Aware Alarm Clock to be able to adjust their wake-up time, based on factors in the environment. In order to achieve this, there are some other requirements which must be fulfilled.

More specifically, for this application, the principal wants to have their clock respond to the weather, and particularly respond to a weather value that is not a value that they want or a weather value that contains particular keyword. This allows the system to respond to a low temperature or an amount of precipitation, or to words in the forecast such as frost or snow.

Further, the principal wants to create rules that allow them to map weather information
into changes of their alarm time. These rules should be composable in such a way that they make sense to the principal’s situation (some rules should override other rules, whereas others should apply together.

For example, the principal might want to have one rule that applies a 15 minute change earlier if there is frost in the overnight forecast, another that applies a 30 minute change earlier if there is 1-3 inches of snow and another that applies a 1 hour change earlier if there is more than 3 inches of snow. If applied together, the principal’s alarm will sound an hour and 45 minutes early, when they would rather only the most severe rule be applied. In a different case the principal might want a warning for heavy rains, and a warning for heavy winds, and for them both to apply when a large storm is forecast.

A further look at how specific requirements for the system might be arranged for a particular principal can be found in Chapter 8. These different requirements will show how specific systems might be constructed to meet the needs of different principals or principals in different situations.

### 7.1.2 Workslip

The workslip for the Aware Alarm Clock has three different types of fields. These are: fields relating to the weather report, fields relating to the weather evaluation and fields relating to the updated alarm time. As there is only one flow of data through the system all of these fields coexist within a single workslip for each instance of the task, which begins with the agents for sourcing weather reports and ends with the agents managing the alarm clock.

The workslip also holds a raw-text representation of the report as a raw element (see Section 4.3). Generally the parsed information was more practical and easily used for agents in the system, and the raw element was not heavily used. For tasks such as these, which are
primarily text based, the raw element may not be needed, but for tasks that include a less easily processed raw element (such as encoded text, or binary data) the raw element may continue to prove useful.

Weather Report Fields

The weather report fields are structured to include common pieces of information that are usually reported regarding the weather. The selection of fields was guided, in particular, by the *Public Weather Forecast Guide* provided by Environment Canada [Env14]. The fields relating to the weather report are shown in Table 7.1.

In Table 7.1 there are 18 different fields included in a workslip. These might not be needed for the alarm clock specifically, but they are the pieces of information that can be retrieved from most weather reports. They are also the pieces of information that a principal can use in conjunction with an evaluation agent to monitor the weather for any concern they have.

Each of these pieces of information is extracted from the weather report (the process for which will be discussed in Section 7.1.3) and inserted into the workslip. This provides a human and machine readable record of the weather that includes the type of information, the potential values and a human readable description. There are a variety of data types and potential values for the 18 different fields. The majority of these are unrestricted Strings, such as the *current condition* and the *forecast*. These fields will primarily be used to look for keywords present or parsed to extract specific values, if their structure allows.

Weather Evaluation Fields

The selection of the weather evaluation fields is based to some extent on the structure of the weather evaluation agents. However, to prevent a chicken-and-egg problem, it is easier to start with the required information, and work towards the structure of the agent that will
### 7.1. AN AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weather report producer</td>
<td>String</td>
<td>Any</td>
<td>The producer of the report</td>
</tr>
<tr>
<td>weather report copyright</td>
<td>String</td>
<td>Any</td>
<td>The copyright of the creator of the report</td>
</tr>
<tr>
<td>weather report current environment</td>
<td>String</td>
<td>Any</td>
<td>The current air quality as evaluated by Environment Canada</td>
</tr>
<tr>
<td>weather report current condition</td>
<td>String</td>
<td>Any</td>
<td>A text description of the current weather condition</td>
</tr>
<tr>
<td>weather report current pressure</td>
<td>Real Number</td>
<td>Non-Negative</td>
<td>The current barometric pressure (in kPa)</td>
</tr>
<tr>
<td>weather report current dewpoint</td>
<td>Integer Number</td>
<td>Any</td>
<td>The current dew point (in °C)</td>
</tr>
<tr>
<td>weather report current humidity</td>
<td>Integer Number</td>
<td>0 - 100</td>
<td>The current relative humidity (in %)</td>
</tr>
<tr>
<td>weather report current observed</td>
<td>String</td>
<td>Any</td>
<td>The location / source of the current report</td>
</tr>
<tr>
<td>weather report current pressure_trend</td>
<td>String</td>
<td>{Rising, Falling}</td>
<td>The change in barometric pressure</td>
</tr>
</tbody>
</table>

¹The Environment Canada Air Quality, is obviously only available for weather reports generated in Canada, but is included here because it was available for the implemented systems.
<table>
<thead>
<tr>
<th>weather report current</th>
<th>Real Number</th>
<th>Any</th>
<th>The current temperature (in °C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report current</td>
<td>Real Number</td>
<td>Any</td>
<td>The current visibility (in km)</td>
</tr>
<tr>
<td>visibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report current</td>
<td>String</td>
<td>Form:</td>
<td>The current direction in 16 point cardinal directions and speed (in km)</td>
</tr>
<tr>
<td>wind</td>
<td></td>
<td>〈direction⟩, Number</td>
<td></td>
</tr>
<tr>
<td>weather report current</td>
<td>Real Number</td>
<td>Any</td>
<td>The current temperature adjusted for wind chill (in °C)</td>
</tr>
<tr>
<td>windChill</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report forecast</td>
<td>String</td>
<td>Any</td>
<td>The forecast for the next several days</td>
</tr>
<tr>
<td>weather report published_time</td>
<td>Time</td>
<td>Any</td>
<td>The time the report was published</td>
</tr>
<tr>
<td>published_time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report recorded_time</td>
<td>Time</td>
<td>Any</td>
<td>The time the report was recorded</td>
</tr>
<tr>
<td>recorded_time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report title</td>
<td>String</td>
<td>Any</td>
<td>The report title</td>
</tr>
<tr>
<td>title</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weather report warnings</td>
<td>String</td>
<td>Any</td>
<td>If there are any watches or warnings for the area</td>
</tr>
<tr>
<td>warnings</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.1. AN AWARE ALARM CLOCK

Produce it.

In this case we know that the principal will require one or more evaluations of the weather. Each of those evaluations will have a name (for example, rain_fall, or wind) and a score, which represents how strongly the principal feels about the severity of that element of weather. To encode this information, the workslip needs one field for each evaluation, named with the given name. The value of this field will be the score for the evaluation. The template for these fields is shown in Table 7.2.

This allows the weather report to be evaluated for many potential different situations and all of the evaluation information be included in the workslip. The principal can inspect the workslip to see which evaluations have been run and the particular severity of each. The actual scale of the score is dependent on the preferences of the principal, so long as it is used consistently among the different evaluations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weather evaluation ⟨eval_name⟩ score</td>
<td>Real Number</td>
<td>Any</td>
<td>Numeric weather evaluation as per ⟨eval_name⟩</td>
</tr>
</tbody>
</table>

Table 7.2: Field templates in the Aware Alarm Clock for the weather evaluation

Table 7.1: Fields in the Aware Alarm Clock for the weather report, types and potential values of the field are based on information from [Env14].
7.1. AN AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weather alarm</td>
<td>Time</td>
<td><em>any</em></td>
<td>Update time for the alarm <em>(alarm_name)</em></td>
</tr>
<tr>
<td><em>(alarm_name)</em></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: The field template in the Aware Alarm Clock for the alarm update

**Alarm Control Fields**

As with the weather evaluation fields, the fields relating to the control of the alarms are based, to some extent, on the structure of the alarm controlling agents. Generally, however the principal wants to be able to send an updated time from a control agent to an agent that will actually sound the alarm. The information needed in this case is simply a time that the alarm should sound. As the weather changes and the control agent updates its decision, it can send an updated value in the field.

The principal may want several different alarms with different criteria. Each alarm is given its own field in the same way each weather evaluation is given its own field. Table 7.3 shows the template of a field for controlling the alarm and its type and potential values.

This format allows alarm sounding agents to check for updates to the alarms they are interested in, and update their internal mechanism correspondingly. For example, an agent on the principal’s smartphone might update the native alarm application with the new time for the alarm to sound. This also allows several different agents to watch for, and update, a given alarm if the principal wants the alarms to sound in different places or ways.

These fields describe the information a principal may need to construct an aware alarm clock. With an understanding of what fields they wish to include, now the principal must decide which agents will best make use of this information to create the system that meets their needs as best as possible.
7.1. AN AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>The name of the agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>A text description of the role the agent plays and its behaviour.</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>A list of the fields the agent requires to operate, or none if the agent will work without any particular field being needed.</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>A list of the fields the agent will update (create or change) after it has executed, or none if the agent will not update any workslip fields.</td>
</tr>
</tbody>
</table>

Table 7.4: The identity information for an Agent

7.1.3 Agents

As discussed, the task of the Aware Alarm Clock can be broken down into four subtasks: retrieving environment data from a source, evaluating that environment data, determining if the alarm should be updated, and managing the alarm. The agents a principal will need to for the Aware Alarm Clock task will match to those four subtasks, and also allow the necessary transformation of information in the workslip.

With an ecosystem of agent providers, a principal would ideally only need to consult their sources of agents, and find agents that match their task, and read and update the necessary fields in the personal agent system. To identify an agent, the agent provider will need to provide a name and a description of the agent as well as a list of fields that the agent requires and a list of fields that the agent updates.\(^2\) Table 7.4 provides an example identity block for an agent.

In effect, the principal is looking for the set of agents which can be used to transform the workslip fields into the desired workslip fields as an outcome. In this case, the principal wants to arrive at the field `weather alarm (alarm_name) time`, which is the field that contains information about the updated time of the alarm. To do that, they will need to find agents

\(^2\)In practice an agent provider will need to provide more information, but this is the minimal set needed for a principal to look at the agents available and determine which ones will be useful.
that produce that field, and then look at the fields those agents require. This means that the principal will want at least one agent to sound the alarm (Alarm Sounding Agents), at least one agent to control the alarm by taking the weather evaluation (Alarm Control Agent), at least one agent to provide an evaluation given the weather report (Weather Evaluation Agents), and at least one agent to generate the weather report (Weather Source Agent). The flow of information is illustrated in Figure 7.1.

This is a very simple flow of information through the personal agent system with information flowing in one direction and all agents in the system needing to touch the workslip.
In this case, the workslip might also exit the system after the Alarm Control Agent has run if it determines that there is no need to update the alarm agent. However this provides a good example of how a principal will need to go about choosing agents for their system.

With no actual ecosystem to source agents from, there are only the four example agents developed for the system, which accurately map to the fields of interest. The remainder of this section introduces the implementation of these agents, and explores their behaviours. The following section includes a description of the specific implementations of the agents and how they are organized using TWPARI-J to create a usable system. In Chapter 8, examples of particular configurations of the alarm clock are explored in depth.

**The Weather Source Agent**

The Weather Source Agent needs to be able to work on the first subtask in the Alarm Clock. It must be able to retrieve weather information from a source and digest it into a format that can be easily used in the workslip. The identification block for the Weather Source Agent is shown in Table 7.5.

The Weather Source Agent does not need to read any data from a workslip to begin working. Following the general pattern of TWPARI-J, the Weather Source Agent only needs to receive an empty workslip to begin working.

Once working, the Weather Source Agent consults its weather source and retrieves a weather report. The agent has to be implemented and configured in such a way that it can find a source and interpret the report. Implementation details will cover the different potential sources of reports (governmental reporting vs particular news outlets) and configuration details will cover different areas of reports (a report for one city or another).

Once retrieved, the report is added to the workslip as its raw element, and then the agent parses the report to extract values to store in each of the fields of the workslip.
Table 7.5: The identity information for the Weather Source Agent

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Weather Source Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Takes a weather report and creates a workslip with the report information</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>none</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>weather report producer, weather report copyright, weather report current environment-CanadaAirQuality, weather report current condition, weather report current pressure, weather report current dewpoint, weather report current humidity, weather report current observed, weather report current pressure_trend, weather report current temperature, weather report current visibility, weather report current wind, weather report current windChill, weather report forecast, weather report published_time, weather report recorded_time, weather report title, weather report warnings</td>
</tr>
</tbody>
</table>

The Weather Evaluation Agent

A Weather Evaluation Agent needs to be able to work on the second subtask in the Alarm Clock. It must be able to evaluate weather information from a workslip, and determine how significant the weather condition is in the report. The identification block for the Weather Evaluation Agent is shown in Table 7.6.

The Weather Evaluation Agents need a workslip containing a weather report to begin working. Once the agent receives a workslip containing the fields it is interested in (the pieces of the weather report it evaluates), then it can process the workslip.

Each agent should have a condition it is looking for, and a method of evaluation that should be configured by the principal when they are setting up the agent. As discussed earlier the principal may want an agent that identifies a keyword or an agent that identifies
<table>
<thead>
<tr>
<th><strong>Agent Name:</strong></th>
<th>Weather Evaluation Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Agent Description:</strong></td>
<td>A configurable agent to evaluate weather conditions</td>
</tr>
<tr>
<td><strong>Fields Read:</strong></td>
<td>weather report producer, weather report copyright, weather report current environment-CanadaAirQuality, weather report current condition, weather report current pressure, weather report current dewpoint, weather report current humidity, weather report current observed, weather report current pressure_trend, weather report current temperature, weather report current visibility, weather report current wind, weather report current windChill, weather report forecast, weather report published_time, weather report recorded_time, weather report title, weather report warnings</td>
</tr>
<tr>
<td><strong>Fields Updated:</strong></td>
<td>weather evaluation (\langle eval_name\rangle) score, weather evaluation (\langle eval_name\rangle) keyword</td>
</tr>
</tbody>
</table>

Table 7.6: The identity information for the Weather Evaluation Agent
7.1. AN AWARE ALARM CLOCK

the severity of a condition in the report.

Four different methods of evaluation are currently implemented for the Aware Alarm Clock. These include:

- A **KeywordEvaluator** - which evaluates if a particular word appears in a given field. It will generate one score if the word is present, and a different score if the word is not present. For example, the agent could test if there was “Frost” in the forecast for the morning.

- A **ValueRangeEvaluator** - which evaluates if the value of a particular field falls in a given range. It generates one score if the values is in the range, and a different score if it does not. For example, the agent could test if the temperature was going to be between $5^\circ C$ and $15^\circ C$.

- A **ValueTableEvaluator** - which evaluates if the value of a particular field falls in a particular range in a given table. It generates its score by looking up the value against the table and taking the value provided by the principal. For example the agent could test if the wind speed was going to be less than $5km/h$, between $5km/h$ and $10km/h$, between $10km/h$ and $20km/h$, or more than $20km/h$ and assign a score of 50, 100, 150, or 200 accordingly.

- A **ValueMappingEvaluator** - which evaluate the value of a particular field by mapping the potential range of the value on the the potential range of scores. It returns a value between the minimum and maximum score, lineally interpolated from the value within its potential range. For example, the agent could map the snowfall of $0cm$ to $30cm$ onto a score of 0 to 200.

This provides a large degree of flexibility to the principal to organize their personal agent
7.1. AN AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Alarm Control Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>An agent that manages alarm times based on environmental evaluation</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>weather evaluation (\langle\text{eval_name}\rangle) score, weather evaluation (\langle\text{eval_name}\rangle) keyword</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>weather alarm (\langle\text{alarm_name}\rangle) time</td>
</tr>
</tbody>
</table>

Table 7.7: The identity information for the Weather Evaluation Agent

systems. The principal will have to configure each evaluation, and particularly will have to determine what scores to assign to the situations the agents will observe.

Whichever method the agent uses to evaluate the particular condition it is evaluating, once it has completed the evaluation, it adds the results to the updated field for the evaluation. The principal will be able to combine evaluations by running multiple agents, each with their own criteria to evaluate and combining their results in the Alarm Control Agent.

The Alarm Control Agent

The Alarm Control Agent needs to be able to work on the third subtask in the Alarm Clock. It must be able to take weather evaluations from a workslip and determine how much, if at all, to change the alarm time by. The identification block for the Alarm Control Agent is shown in Table 7.7.

The Alarm Control Agent needs evaluation fields in order to work. It is possible to have several different Alarm Control Agents that each monitor a particular aspect of the weather and act accordingly, but this limits the potential for the principal to combine their evaluations in complex ways - ”Wake me up 30 minutes earlier if there’s frost, but if there is snow, then leave my alarm alone as I won’t be going in”. This is an instance of the application of the guideline that agents should be “As simple as possible, but no simpler,” as making the Alarm Control Agent simpler makes it harder to use, and might produce race
### 7.1. AN AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Alarm Name:</th>
<th>Snow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Alarm Time:</td>
<td>0700h</td>
</tr>
<tr>
<td>Alarm Update Table:</td>
<td>weather.evalution.snow.score 100 -15</td>
</tr>
<tr>
<td></td>
<td>weather.evalution.snow.score 300 -60</td>
</tr>
<tr>
<td></td>
<td>weather.evalution.total.score 100 -30</td>
</tr>
</tbody>
</table>

Table 7.8: The alarm update table for an example alarm for a principal

conditions in the update of the alarm and other unwanted side effects.

After it receives a workslip with evaluation information, the Alarm Control Agent can either make an adjustment to the alarm immediately, or it can wait for a specified period of time to aggregate several evaluations, and adjust the alarm based on that aggregation. This can be adjusted, depending on the environment in which the principal is running the agents, as it is possible that the principal may want to limit the number of messages sent, or ensure that the alarm is as up-to-date as possible (which, given the nature of the problem is not always particularly necessary).

The Alarm Control Agent can keep track of several alarms and the conditions under which they should be updated (the *Alarm Update Table*, see table 7.8 for an example). For each alarm, it keeps track of the original time the alarm was set for, and then all of the conditions that it should be applying. In this way it can combine conditions intelligently, as discussed above, to produce the current time that the principal would like the alarm to sound. As it updates its alarms, it updates a workslip field identified with the name of that alarm with the new time at which it should sound and then sends that workslip onward for Alarm Sounding Agents to read.
7.1. AN AWARE ALARM CLOCK

The Alarm Sounding Agent

An Alarm Sounding Agent needs to be able to work on the fourth subtask in the Alarm Clock. It must be able to take updated alarm times and use them to update its own internal alarm sounding mechanism. This might be an internal mechanism, an update to another program or even a system of logging rather than actual sounding. The identification block for the Alarm Sounding Agent is shown in Table 7.9.

The Alarm Sounding Agent waits for workslips with a field corresponding to an alarm it keeps. When it receives that update, it adjusts the time of the alarm to the newly updated time (and sounds the alarm if necessary, if the new alarm time is now in the past). After doing its work, the Alarm Sounding Agent does not need to update any further information for other agents, so it does not update any fields nor produce a workslip.

The Alarm Sounding Agent is one which will be most personal to the principal. Depending on the system the principal uses to keep their alarms they will need a different type of Alarm Sounding Agent to manage that system. Alarm Sounding Agents can be custom programmed for different platforms or they can be designed to integrate with existing systems.

With these four types of agents, it is now possible to look at how a personal agent system built from these agents might work. First it is important to see how the principal can organize, and arrange the agents, then to see how the workslips will flow through the agents to produce intelligent alarm updates.

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Alarm Sounding Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>An agent that updates and sounds an alarm</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>weather alarm (\text{alarm_name}) time</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>\textit{none}</td>
</tr>
</tbody>
</table>

Table 7.9: The identity information for the Weather Evaluation Agent
7.1.4 System Application

The principal has already completed the first two tasks when setting up a personal agent system using the Task and Workslip Scheme. They have identified the fields they will need to get the task they want done and they have identified the agents that will transform those fields to produce the outcome that they want. The remaining tasks are to setup and configure the agents, and to allow the system to run.

Each of the agents are implemented generally as described in the previous section, with some specific details, discussed below and each agent must be configured to reflect the principal’s plan for the task:

- The Weather Source Agent is specifically implemented to retrieve an Environment Canada Weather Forecast using RSS. It is configured with the feed url for a particular region of Canada.

- The Weather Evaluation Agents are configured to look for particular fields and configured to respond to appropriate weather, depending on the principal’s wishes.

- The Alarm Control Agent is configured with the alarms the principal wants and the table showing how those alarms are updated according to the evaluations from the Weather Evaluation Agents.

- In the examples shown in Chapter 8, two implementations of the Alarm Sounding Agent agents are used: the Simple *nix Alarm Agent, which sounds an alarm from the *nix command line, and the Simple Logging Alarm Agent, which instead records the time the alarm would sound at in a log (along with information about how it was updated).
7.1. AN AWARE ALARM CLOCK

The Alarm Sounding Agents are configured with the alarm they are sounding (or logging).

Abstractly, setting up the agents is relatively simple, the principal only needs to add them to their execution environment and the agents will communicate all of their workslips and the principal’s alarm will be updated. In practice a little more coordination is required. Following the approach in Chapter 6, a Dispatcher will be used to manage the movement of workslips. Agents will run in an environment where they will be able to share files with the Dispatcher and as they update workslips they will pass those files to the dispatcher, which will then look at the updated workslips fields and compare it to its list of fields read for each agent and send the workslip on to those interested agents. This requires that as agents are activated, they must first register their list of workslip fields they read with the dispatcher (and the location of their input directory).

The final practical consideration is how to have the system run at an appropriate rate. Weather reports are only updated on a periodic basis, so the Weather Source Agent only needs to execute from time to time. To achieve this a script is used that periodically pushes an empty workslip to the Weather Source Agent. When that agent receives the workslip it begins to work, updating weather report fields, so the workslip can then be passed to the Weather Evaluation Agents via the Dispatcher.

With the workslip fields and agents selected and the agents deployed to the principal’s environment, the principal can now use their new personal agent system to wake them up at the best possible time. The system is able to respond to the weather, tracking scalar or keyword values and use those in a coherent way (according to the wishes of the principal). As mentioned, examples of how particular principals can take these agents and this system and use it effectively are shown as part of the evaluation Chapter 8.
Beyond those evaluations of the effectiveness of the system for a principal, it is also worth comparing the task and the system to the philosophical guidelines in Chapter 5. As already discussed, the task itself is well suited to the Task and Workslip Scheme, being regular, repeated and informatic.

The agents are well suited because they have been implemented in a way that is as simple as possible without being too simple. As can be seen, each agent in the system does one task step in as limited a way as makes sense for the principal. The Weather Source Agents consult a single weather report. The Weather Evaluation Agents checks a single aspect of that report. The Alarm Control Agent combines evaluations and updates alarms. The Alarm Sounding Agents manage the actual act of sounding the alarm.

This particular task does not illustrate the reusability of the agents very well. Multiple instances of the agents are useful to handle things such as different weather evaluations, or different alarms, but the intention is to see that agents are reusable between systems or in different roles in a system. It also would not be difficult for the principal to add new agents looking at different weather sources, or even different environmental aspects (such as traffic) to augment the system.

Finally, agents should be clear in their requirements and productions: the TWPARI-J system makes this easily achieved, as each agent shares the workslip fields it is interested in via the Dispatcher. The Agent Information Blocks also make it simple for the principal to evaluate an agent and determine if the agent meets their needs and fits into the system they are constructing.

The workslip is also well suited to the task. On the one hand this is due to the TWPARI-J implementation following the guidelines, but additionally this can be seen in the role of the workslip in the system. It is simple for agents to pass workslips, and the AgentCell allows someone implementing the agent to easily read and send a workslip without needing to know
any of the details about how the agent will send this information.

The workslip provides the history of the system on its surface. Fields are updated by each agent as they go past and no agent updates a field updated by another agent. This provides a simple illustration of the history keeping role of the workslip.

The workslip makes it easy to provide extra information about the forecast, and allows the evaluation agents to only concern themselves with a particular value (of a particular type) in a particular field, rather than having to parse that information separately. In the event that the workslip field updates provided by the Weather Source Agent do not adequately provide the information an agent is looking for, the raw element allows the agent to retrieve that information if necessary. This would require a different implementation, or better, an alternate agent that could also be added to the system with the different evaluation agent.

Finally, the alarm clock is a good example of the guidelines for personal agent systems. *The system is personal.* This is due to the configurability of the agents and the configurability of the system by composing the agents. The principal can select their preferred weather source, evaluate the weather in the way they want and combine those evaluations into effective rules for them.

*The system is automatic.* The principal only needs to ensure that the Weather Source Agent is checking the weather regularly (as prompted by receiving an empty workslip). After that the agents all receive the workslips and with the Dispatcher, will only run when a workslip with fields they are interested in is available to work on.

Finally, *the system is efficient and discreet.* None of the steps in the system take significant computational power, and additionally this system will be primarily active when the principal is asleep. The weather report is unlikely to change more than a few times an hour, even at its most active, and as such the whole system will only need to run a few times an hour.
Combined, these factors show that this a good problem for the Task and Workslip Scheme for Personal Agents. Further, although this is the simplest example, it provides a clear look at the role agents and workslips play in a Personal Agent System developed under the scheme. The next example system, the shopping list, provides examples of how agents can combine information from different sources into a coherent and helpful set of information.

7.2 An Efficient and Price Aware Shopping List

As with the Aware Alarm Clock, the Aware Shopping List is based on the example problem outlined in Section 2.2. In it, the Aware Shopping List task is to minimize overall shopping cost and time (in some combination) by linking the principal’s shopping list to sale information from various stores. For example, a principal might want to save as much as they can on paper towels, so they will travel to any store to get them at the best price, however they may only like to purchase oranges at one particular store or they may not care where they purchase milk at all. A principal will have a list of items they need, and requirements on those items such as where they would like to get them from, or a price they would like to get them at. Agents gather information from stores and combine them into a useful format, which can then be compared against the shopping list and the requirements to produce a plan for the principal to go shopping with.

There are four subtasks for this task: first, a shopping list has to be maintained for the principal in a format that both the principal and agents can work with, second, price and sale information has to be obtained from the various stores that the principal would like to shop at. Third, the shopping list and the sales information needs to be combined into a useable plan for the principal. Finally, that plan has to be communicated to the principal.

As a task, the Aware Shopping List is well suited to the Task and Workslip Scheme,
and as with the Aware Alarm Clock, it demonstrates the three attributes for tasks in the philosophical guidelines. It is repeated, regular and informatic. The task is repeated because this is a task that the principal will want to run on a regular basis, once or maybe twice a day. The task is regular, because the output of the system is a shopping plan, including a list of store locations and for each, a list of items which should be purchased there. Finally, the task is informatic, because all of the parts of the task are easily managed by a computer, from gathering information from a web source, to gathering a list from a principal to comparing the two sources of information to generate a plan.

The system presented here is an adaptation of an early system built by Mark Barley to work with the Task and Workslip Scheme presented in [Bar16].

7.2.1 Requirements

A principal using the Aware Shopping List system wants to be able to get a shopping plan that suits their preferences when shopping, be that lowest price, shortest travel, best quality or some other criteria.

The principal should be able to specify sources to gather shopping information from and for each source there should be a system to convert that source’s information organization into an internal organization that can be stored in a workslip and easily used to produce the shopping plan.

For both the shopping information sources and the shopping list, it is useful to have a item / brand set. That way a principal can specify that they would like to buy “Chicken Soup” and another can specify “Smith’s Chicken Soup”. For the first principal if “George’s Chicken Soup” is cheaper it can be added to the list instead of “Smith’s”. For the second principal the only chicken soup added to the list would be “Smith’s”, regardless of the cost differences
for the other soups. Alternately the first principal could also specify that Smith’s is the preference, but if George’s is significantly cheaper, then George’s should be selected. For the purposes of this example the lists are limited to just a simple item / brand relationship, but in other situations this could be expanded to a much more complex system of related and substitutable items.

In Chapter 2.2 Alice also excluded some brands from a the shopping list, as negative examples. This has not been included here to simplify the general structure, but could be added with a more complete pattern matching system for items and brands.

The principal will want to be able to specify a general best practice for each shopping plan, fewest stops, cheapest groceries, best matches (fewest categories replacing items). Additionally, the principal should be able to specify for each item the extent to which they care if the item is replaced with another and the extent to which they care where they purchase the item.

7.2.2 Workslip

The workslip for the Aware Shopping List System has three different types of fields. These fields are: those related to the principal’s shopping list, those related to the shopping source information and those related to the shopping plan.

For the shopping list, there are very few actual fields, but the fields themselves hold fairly complex information. The shopping list, for example, is contained as a single field with all of the current items on the principal’s shopping list held together as comma separated strings. Similarly, the shopping information from the stores is held with one field per store, but that field contains the current prices for the all items the store sells (that the principal is interested in).
### The Shopping List Field

The shopping list field contains all of the items that the principal would like to buy on their next shopping trip. The template for the field is shown in Table 7.10.

The field is expected to hold a list of items in the form \( \langle \text{item} : \text{brand} \rangle \), where either the item or the brand can be omitted. If only the item is present the later agents must assume that the principal only cares generically about an item (for example, `toothpaste:`). If the item is omitted then later agents must assume that the principal only wants the specific item provided (for example `Shine Brand Toothpaste`). If both the item and the brand are included, other agents can use the principal’s preferences to determine whether to use the item or the brand.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shopping list</td>
<td>String</td>
<td>Form: <code>Shopping List</code></td>
<td>The items the principal would like to shop for</td>
</tr>
</tbody>
</table>
7.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>store ⟨store⟩ sale</td>
<td>String</td>
<td>Form: Sale List</td>
<td>The items available from the store at the given price per unit</td>
</tr>
</tbody>
</table>

Table 7.11: The items available from the store ⟨store⟩

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shopping plan</td>
<td>String</td>
<td>Form: Shopping Plan</td>
<td>List of stores and the items to purchase at them</td>
</tr>
</tbody>
</table>

Table 7.12: The items available from the store ⟨store⟩

The Store List Field

Each store that the principal configures their system to manage has a list of the items available for sale. This information is shared in a field for each store. The template for the field is shown in Table 7.11.

The list is structured in the form: \(c_1 : i_1 : \text{ppu}_1, c_2 : i_2 : \text{ppu}_2, \ldots\) a sequence of triples in the form \(i_{item} : \text{brand}_{i} : \text{priceperunit}_{i}\) which includes the item, brand and the price per unit for each item the store sells. This provides item and brand that can be matched against the principal’s shopping list and a price per unit which provides a common cost for later agents to use to evaluate items against the principal’s preferences.

The Shopping Plan Field

The Shopping Plan field includes the plan produced for the principal to follow when shopping. The template for the field is shown in Table 7.12.

The field contains a plan in the form of a list of stores, each with a list of the items \(\text{store}_1\{i_{item1,1}, \ldots, i_{item1,n_1}\}, \ldots, \text{store}_m\{i_{itemm,1}, \ldots, i_{itemm,n_m}\}\). This provides the list of
items that the principal will want to purchase at each store.

7.2.3 Agents

As discussed, the Aware Shopping List has four subtasks that need to be managed, maintaining the principal’s shopping list, obtaining price information from the stores, combining the list and price information into a shopping plan and communicating that plan with the principal. The principal will need to find agents that are able to accomplish each of these subtasks.

As with the Aware Alarm Clock, an ecosystem of agent providers would ideally provide the principal with agents that not only accomplish the subtask in general, but also specialized to accomplish specific types of the subtask such as different agents to get price information from different stores or different agents to keep the principal’s shopping list in different ways. As always, the principal will need to identify agents by the fields read and updated, as well as by the name and agent description. (Table 7.4 provides an example id block for an agent).

Instead of the generally linear flow of the Aware Alarm Clock, these agents will produce information from different sources and then combine them. However the overall concept is the same, to transform a set of input workslip fields to produce a set of desired output fields. In this case the sought field is shopping plan. This can be produced by a Shopping Plan Agent, which will require several different instances of the field `store ⟨store⟩ sale` from different stores and at least one instance of the field shopping list. These will in turn require several Store Information Agents (each for a different store) and (at least one) Shopping List Agent. Figure 7.2 provides an illustration of the flow of information.

One thing that is important to note is that there will be several different store agents, and possibly several list agents which provide information to the plan agent.
7.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

The workslip flow of the Aware Shopping List, showing the flow of fields of interest from the Store Information Agent(s) and the Shopping List Agent(s), to the Plan Agent and finally to the Principal Communication Agent. The agents are shown as clouds because there may be more than one agent needed.

Figure 7.2

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Shopping List Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Records and shares a principal’s shopping list</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>none</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>shopping list</td>
</tr>
</tbody>
</table>

Table 7.13: The identity information for the Shopping List Agent

The Shopping List Agent

The Shopping List Agent needs to be able to work on the shopping list subtask. It must be able to allow the principal to record their shopping list information, and share that information with other agents. The identification block for the Shopping List Agent is shown in Table 7.13.

The Shopping List Agent does not need to read any data from a workslip to begin working. Following the general pattern of TWPARI-J, the agent only needs to receive any
workslip to begin working.

Once working, the Shopping List Agent consults its stored shopping list for the principal. The exact way the principal’s list is stored depends on the implementation of the agent, but if the agent has an interactive component to work with the principal, that component needs to be outside the “agent” operation, and provide the information for when the agent is ready to activate. When the agent activates, it can parse the list and ensure that items are in correct \(<item, brand>\) form.

Once retrieved, the shopping list is added to the workslip in the shopping list field and the agent can then send the workslip on.

**The Store Information Agent**

The Store Information Agent needs to be able to work on the store data subtask. It must be able to consult with the online information shared by a store and parse it into a format that a Shopping Plan agent can use. The identification block for the Shopping List Agent is shown in Table 7.14.
7.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Shopping Plan Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Constructs a plan of which items a principal should purchase at which stores</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>shopping list, store (\text{store}) sale</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>shopping plan</td>
</tr>
</tbody>
</table>

Table 7.15: The identity information for the Store Information Agent

As with the Shopping List Agent, the Store Information Agent does not need to read any data from a workslip to begin working. Again, following the general pattern of TWPARI-J, it only needs to receive any workslip to begin working.

Once working, the Store Information Agent consults its resource for its given store. This will vary with the different ways stores share information and may involve access to information shared in a structured format or extracting information from an unstructured source. Once the data has been collected it can be formatted into the Sale List Format, including the item, brand and price per unit for each item.

Once retrieved and parsed, the store information is added to the workslip in the store \(\text{store}\) sale field and the agent can then send the workslip on to the Shopping Plan Agent to evaluate.

The Shopping Plan Agent

The Shopping Plan Agent needs to be able to work on the shopping plan subtask. It must be able to take information about what the principal would like to buy (the shopping list), what the stores have available, and at what price, (the store sale information), and combine it into a plan that best suits the principal (based the principal’s configuration). The identification block for the Shopping List Agent is shown in Table 7.15.

The Shopping Plan Agent keeps a current list of items it has seen that the principal would like to buy (a history of the shopping list) and a current set of lists of items that are
### 7.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Principal Communication Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Communicates the shopping plan to the principal.</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>shopping plan</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 7.16: The identity information for the Principal Communication Agent

available from the stores the Store Information Agents are sending. Each time it receives a workslip it updates its current information and then produces a plan according to the principal’s preferences.

The Shopping Plan Agent reads through the information it currently has and assigns each item (or brand) from the principal’s shopping list to one of the stores for which information is provided. The algorithm the agent will use will depend on the principal’s preferences, so if the principal wants the lowest price possible, then items will be assigned according to the lowest price per unit (PPU), whereas if the principal wants the fewest stops possible, then the agent attempts to assign the items to the stores at which the principal already has to stop, even if that causes the price to be higher. Different algorithms can be encoded as different modules for the agent to execute.

Once the plan has been developed, it is added to the workslip in the **shopping plan** field and the agent can then send the workslip on.

**The Principal Communication Agent**

The Principal Communication Agent needs to be able to work on the subtask of communicating with the principal. It must be able to take information about the shopping plan and present it to the principal in a useful and coherent format, using an appropriate channel. The identification block for the Principal Communication Agent is shown in Table 7.16.
7.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

The Principal Communication Agent takes the plan as produced by the Shopping Plan Agent and then communicates it to the principal. There are many different forms that the agent could take, such as to present a webpage to the principal, or to connect with an app on their phone, or to send an e-mail to the principal. Different versions of the agent can communicate the list to the principal in the way the principal feels is most useful. As with the Alarm Sounding Agent, the principal will experience this agent most directly and to some extent this agent will seem to be the whole of the system.

Once the Principal Communication Agent has sent the plan to the principal, then there is no further work for the system to do. In practice the agents will wait for another update to the shopping list or to the store information and then the plan agent will generate and send a new plan.

As with the Aware Alarm Clock, now that the agents have been discussed it is possible to look at the organization of a personal agent system using these agents to help a principal organize their shopping list.

### 7.2.4 System Application

The design of a personal agent system is predominantly managed through the intended flow of the workslip and the agents needed to make the necessary transformations to the workslip to produce the result the principal was hoping for. The agents outlined above provide the necessary transformations, but for the application of the system, there need to be some specific implementation details.

- The Shopping List Agent is specifically implemented to receive the shopping list from the principal via a text file.
• The Store Information Agents were originally designed to collect public information shared on the website of a particular chain of stores.

• The Shopping Plan Agent is designed to match items from the shopping list to the stores provided and can load different modules to do the actual generation of the plan.

• The Principal Communication Agent is designed to produce a text file with the principal’s shopping plan.

As with the Aware Alarm Clock, a Dispatcher is used to coordinate the passing of workslips between agents. Each agent registers with the Dispatcher and provides the fields it is interested in and the Dispatcher then forwards new workslips to interested agents as they become available. Again the agents here are run in an environment where they are able to communicate by passing files from one agent to another.

The Aware Shopping List does not need to run as frequently as the Aware Alarm clock, generally the store information will only be updated once a day (if that) and the shopping list will only be updated a few times a day. Having the agents run on a daily basis would be reasonable, although that there should also be a way for the principal to run the system to produce a plan when they are ready to go shopping. In this case all a principal will need to do is update the shopping list which will push a workslip through the system causing the plan to be generated and sent back to the principal.

With this organization the Aware Shopping list is sufficiently effective for a principal to use to manage their shopping. Again, as with the Aware Alarm Clock, the task and system should be evaluated with its relation to the Philosophical Guidelines for the Task and Workslip Scheme, discussed in Chapter 5.

As already discussed, the task itself is well suited to the Task and Workslip Scheme, being regular, repeated and informatic.
The agents in the system also hold to the guidelines. While the agents are more complicated than those of the Aware Alarm Clock, they do each do one task (subtask). The Shopping List Agent maintains one list. Each Store Information Agent is able to retrieve information from one store, although they could conceivably be broken down into two agents, one to retrieve the information and one to parse it. In this case it was decided to leave them as a single agent because the information being retrieved is in a format that is highly tied to its location (that is to say either repeated requests are needed or that simply for each store there is a wholly different structure for the information). The Shopping Plan Agent only combines shopping list information and store information to produce a plan, and the Principal Communication Agent only needs to send a plan to the principal.

As with the Aware Alarm Clock, the reusability of the agents is not highly demonstrated in this system. There are many systems that could take further advantage of these agents (especially the Principal Communication Agent, but also the Shopping List Agent and the Store Information Agent). Similarly, it would be conceivable to combine the Weather Source Agent and Weather Evaluation Agent of the Aware Alarm Clock with this system, which could provide different shopping plans depending on the weather.

And again, as with the Aware Alarm Clock, TWPARI-J makes the agents clear in their requirements and productions. Every agent is able to express the fields it will need to read in order to work and the fields it will update. This, combined with the plain text description of the agent’s operation in the Agent Information Block, allows a principal to determine if the agent is able to work in the system they are planning.

The workslip for the task is also well suited according to the philosophical guidelines. Again, TWPARI-J provides a structure for agents to share workslips, and interpret workslips in the same fashion. The history of the workslip is managed via having each agent update a unique field and the history of the updates to be built in the Shopping Plan Agent. This
allows a principal to see the decision making process of the system over a single instance of the system running, and to see over several instances the full history in the knowledge built in the Shopping Plan Agent.

One of the places where the Aware Shopping List differs significantly from the Aware Alarm Clock is in the structure of the fields in the workslip. For the alarm clock, each conceptual item (such as a weather value or evaluation) is an entry in a separate field. For the shopping list, each conceptual item is part of a complex data structure within a field. Both of these systems work as demonstrations of the Task and Workslip Scheme, but the shopping list requires agents to have a closer agreement on the data being shared (particularly its structure).

The complexity of the data structure limits the reusability of the agents because agents will have to agree upon how the data is organized. However, while this does not break the concepts of the Task and Workslip Scheme, it is still entirely possible for agents to work together sharing information with the fields describing shopping lists, store information and shopping plans. In this case, this can be used as an illustration of the range of data complexity that the Task and Workslip Scheme can sustain. Generally less complicated workslip fields should be preferred, but in situations where complexity cannot be avoided, the agents are still able to cooperate via complex data structures (at the cost of reusability).

Finally, the Aware Shopping List is another good example of the guidelines for personal agent systems, because it is personal, automatic and efficient. The personal aspect of the shopping list is clear, each principal has their own preferred items when shopping, their own preferred stores and their own preferred style for getting the most out of their shopping trips. The system allows each of these to be configured and changed as the principal changes their opinions.

The automatic aspect of the shopping list is also clear. As explored in the motivating
story of Alice and Bob in Chapter 2, for a person this task can become very complicated and time consuming, but can be automated with relative ease. The sticking point for automation is the cooperation of stores and the availability of their shopping information, but as seen in [Bar16] if the information is provided in a usable format, agents can be developed to work with it.

Finally the system is highly efficient and discreet. The timeframe the agents need to operate in is quite slow from a human perspective and the overall computational load of any agent in the system is quite low. The Principal Communication Agent provides the principal control over their preferred form of communication, but low impact methods such as e-mail make it fairly easy for the system to limit intrusion into the principal’s life.

Across all of these factors, the Efficient and Aware Shopping List is another task and system well suited to the Task and Workslip Scheme for Personal Agents. It provides an example of how data from different sources can be combined into a useful result. It is also an example of how the scheme can allow flexibility in the design of workslips and the structure of workslip fields.

The first two example systems have looked at personal tasks that occur on a human time scale. The final example system will demonstrate how agents can be used to provide proactive assistance in timeframes that humans cannot directly respond to and how agents can handle multiple workflows for information depending on the situation surrounding a task.

### 7.3 An Informative and Proactive System Operator

The final example task is drawn from the concept in Section 2.3. Here the Proactive System Operator (PSO) supports a principal working as technical support. The principal would like
to know as soon as possible when a service has stopped working. Ideally they would like to have the system react to the problem, and repair the problem if possible.

The term *service* is used here to indicate a computer system outside of the personal agent system. Examples of services include a principal’s SSH service, or their Web Hosting service. Any system that the personal agent system is monitoring is a *service*. Any aspect of the principal’s personal agent system will be called a system.

This task can be broken down into four subtasks: identifying a service’s status, reacting to a service not functioning correctly, tracking a service’s status and alerting a principal about events.

As a task, the Proactive System Operator is well suited to the Task and Workslip Scheme. It illustrates each of the philosophical guidelines for tasks, being repeated, regular and informatic. The task is repeated because the personal agent system must check each of the services that the principal is concerned with. Unlike for the other example systems where the repeated time frame might be hourly or daily, the frequency comes closer to by-the-minute or even more frequent. It is not at such a frequency that specialized software is necessary.

The task is regular because for each service that the personal agent system interacts with there is a limited set of states the service can be in (functioning, some kind of error state or not accessible) and a limited set of actions that can be taken to restore those services. Within these constraints the information and action of the system must be finite and as such the goal state of the system must be reachable.

Finally the task is informatic because all of the information that the system needs to gather and produce, and all of the actions the system can take are easily managed by computer.
7.3.1 Requirements

The principal for the Proactive System Operator will require the system to be able to notice all failures of the services it is set to monitor, and to react to and attempt to recover those that it is able to recover. To do this, the principal needs to be able to configure agents to correctly identify a service that is not functioning correctly and to configure agents to correctly react to those services.

The configuration of the system allows the principal to determine how events are managed. It allows them to stay informed of all problems, possibly with high-importance information being sent faster or more noticeably than low-importance information. It allows the principal the flexibility to allow reactions to be automated, either in the case where the reaction should be done faster than the principal can reliably react, or for situations where the reaction is possible without the principal’s immediate participation. In either case the principal will always be able to stay up-to-date with information from the system and so if a reaction does not succeed properly the principal will be able to react themselves.

7.3.2 Workslip

The workslip for the Proactive System Operator has three different types of fields. The first type of fields relates to the status of individual systems. The second type of fields are related to agents’ efforts to react to systems not functioning, and finally there are workslip fields that reflect the overall state of all the systems the PSO is monitoring.

Service Status Fields

The service status fields include information about the status of services. At the most abstract they include a field to report on the condition of a system (function, in an error
7.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Potential Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pso (service_name) status</td>
<td>String</td>
<td>UP, DOWN, ERROR</td>
<td>Current Status of service (service_name)</td>
</tr>
<tr>
<td>pso (service_name) error</td>
<td>String</td>
<td>Any</td>
<td>Error condition of the service (service_name) (if retrievable and applicable)</td>
</tr>
</tbody>
</table>

Table 7.17: The field template for the Proactive System Operator (PSO) for the system status fields.

state or unreachable) and a field a to report supplemental information regarding any errors (if there are any). These two fields are described in Table 7.17. Both of these fields provide information that the system may use to react to an event or to pass on to the principal instead.

The field pso (service_name) status should be updated every time an agent checks the status of a system. This field allows values of UP, DOWN, and ERROR. If the service is accessed successfully, then the field should be updated with the UP status. If the service cannot be accessed at all, then the field should be updated with the DOWN status. If the service can be accessed but is not working as expected then the field should be updated with the ERROR status.

The other field, pso (service_name) error, should only be updated if the service is in the error state. The field provides supplementary information that could help recover the service later.

**Service Reaction Fields**

The service reaction fields include information about the work agents have done in reaction to a change in status of a service. The reactions can cover a wide variety of different actions
including attempts to bring the system up, active a back up system, send an alert (to people other than the principal) or to change some other system configuration to improve the situation. These two fields are described in Table 7.18.

The potential reactions to a service being down are very broad. To account for this, the fields allow a broad array of information to be recorded. The field `pso ⟨service_name⟩ reaction ⟨reaction_name⟩` allows an announcement that the reaction has taken place, which may allow further reactions by other agents in the system or may simply be informative for the principal. The second field `pso ⟨service_name⟩ reaction ⟨reaction_name⟩ status` is used to provide supplemental information about the reaction which again may be used by other agents or by the principal.

For example supposing that an FTP server were to stop working. Initially the workslip field `pso FTP status` would be updated in a workslip which is passed on to other agents one agent might be configured to update a status page and so would update the field `pso FTP reaction update page` that the attempt was made. If the agent was for some reason unable to access the status page then it might update the field `pso ⟨service_name⟩ reaction ⟨reaction_name⟩ status` that the attempt to react had failed.
### State Update Fields

The state update fields provide information about changes to all of the services that the PSO is monitoring. This includes information gathered about the status of individual services and about attempts to react. In effect these fields provide a digest of information intended for the principal regarding the overall state of the services. These fields are described in Table 7.19.

The field **pso state update** provides information about a service that the system is watching. Generally it is intended to update the principal about changes in the services such as a service going down, or being fixed. As such a normal value for the field might be *FTP:stopped responding* or *ft:started responding*. The messages are intended to be human readable, and informative, for the principal.

The field **pso state priority** provides a priority level which can be used by agents to determine how to correctly communicate with the principal. The principal may want to configure the system to simply log all low level messages, receive e-mails for regular or high ones and receive a text message for ULTRA priority messages. In any case the system should
allow the principal to configure the correct priority levels for different situations.

The field **psos state details** provides supplemental information if available and configured by the principal. Supplemental information might include information extracted from the extra details that were included in the error information from the service status, from attempts to react, or other information such as the length of time the service has been in its current condition.

### 7.3.3 Agents

There are four subtasks identified for the Proactive System Operator task: identifying a service’s status, reacting to a service not functioning correctly, tracking a service’s status, and alerting a principal about events. Each of these can easily be assigned to an agent.

As with the first two example systems, an ecosystem of agents and agent providers should be able to provide a principal with agents that can complete each subtask. Again a variety of agents would allow for the various types of monitoring, and response, that a principal might want for their system, and a principal would only need to find agents that fulfill the task requirements in the way they want.

The flow of workslips for the PSO is different then the flows seen for the Alarm Clock or the Shopping List. As illustrated in Figure 7.3, the service status workslip fields from the Service Testing Agents are of interest to both the Service Response Agents and the System Status Agents. If the Service Reaction Agents read a status in the service status fields and have a reaction, then they will also produce relevant service reaction fields. The System Status Agent will read service status fields, and service reaction fields, and will use them to construct a model of the state of the services and write state update fields. The Principal Communication Agents will read the state update fields and then appropriately send them
Figure 7.3: The workslip flow of the Proactive System Operator, showing the flow of fields of interest from the Service Testing Agents (STAs), to the Service Reaction Agents (SRAs) and to the System Status Agent (SSA), Principal Communication Agents(s). The agents are shown as clouds because there may be numerous agents, but each type of agent will participate in this flow.

to the principal.

As with the shopping list system, it is worth noting that there will be several different agents, especially those testing services and reacting to services. All of these will feed into the System Status Agent which will use their information to form its model of the system.

**Service Testing Agent**

Service Testing Agents work on the subtask of testing the status of a service, and, if possible, determining what the problem is with a service. Each agent must be able to be configured to be able to test a particular service and send that information on for other agents or the principal to use. The identification block for a Service Testing Agent is shown in Table 7.20.

A Service Testing Agent does not need to read any information from a workslip to begin
7.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

**Agent Name:** Service Testing Agent

**Agent Description:** Tests a service to ascertain what its status is and if it is experiencing any errors

**Fields Read:** none

**Fields Updated:** \( pso \langle \text{service\_name} \rangle \text{status}, pso \langle \text{service\_name} \rangle \text{error} \)

Table 7.20: The identity information for a Service Testing Agent

work. As with other such agents, and following the pattern of TWPARI-J, the agent only needs to receive a workslip.

Once working the Service Testing Agent tests the service it is configured to monitor. The exact nature of the test is dependent on the service, the agent and the preferences of the principal. Generally the agent can expect to be able to determine if the system is functioning normally, if it has some kind of error or if it is not functioning at all.

For example if testing a web service, the agent may note that the service is correctly returning pages, or it may note that errors are being returned or it may find that nothing is being returned. In each of these cases the agent will respectively record that the service is \textit{UP}, \textit{ERROR} and the type of error recorded in the error field, or \textit{DOWN}.

The agent packages the results of the test into the workslip fields \( pso \langle \text{service\_name} \rangle \text{status} \) and \( pso \langle \text{service\_name} \rangle \text{error} \) (if necessary), and sends the workslip on.

**Service Reaction Agent**

Service Reaction Agents work on the subtask of reacting to a service in a particular condition. Each agent must be able to be configured to be able to react to a situation in the way that the principal wants and must be able to inform the principal and other agents about the reactions it is taking. The identification block for a Service Reaction Agent is shown in Table 7.21.
### 7.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>Service Reaction Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Reacts to a particular status of a service</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>( pso{}^{}(service_name)) status, ( pso{}^{}(service_name)) error</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>( pso{}^{}(service_name)) reaction ( reaction_name ), ( pso{}^{}(service_name)) reaction ( reaction_name ) status</td>
</tr>
</tbody>
</table>

Table 7.21: The identity information for a Service Reaction Agent

A Service Reaction Agent looks for information it needs in the \( pso{}^{}(service\_name) \) status and \( pso{}^{}(service\_name) \) error fields. Generally it will either look for a particular status or a change in status. Different agents will be necessary to handle different reactions and different reactions will be needed for different services at different times depending on the needs of the principal. Generally the agent will attempt to do something that the principal wants when the condition for the action is met.

For example, a reaction might be to attempt to restart a service that stopped working. The Reaction Agent would notice that the service had gone from \textit{UP} to \textit{DOWN}, and it could then execute a script to restart the service. If the script executed and exited normally, the Reaction Agent would record that the attempt was successful (whether or not the service was actually restored by the attempt, the attempt itself succeeded). Alternately if the script exited unexpectedly then the Reaction Agent would record that the attempt had failed.

Other reactions an agent might have is to bring up a backup service if a service had gone down, to change a configuration if the service was returning an error, or to send a message to another service to let a user of the service know it is down and that actions are being taken.

Whatever the result of the reaction the agent records the attempt and the result into the \( pso{}^{}(service\_name) \) reaction \( reaction\_name \) and \( pso{}^{}(service\_name) \) reaction \( reaction\_name \)
### 7.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Agent Name:</th>
<th>System Status Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Description:</td>
<td>Reacts to a particular status of a service</td>
</tr>
<tr>
<td>Fields Read:</td>
<td>$pso \langle service_name \rangle$ status, $pso \langle service_name \rangle$ error, $pso \langle service_name \rangle$ reaction $\langle reaction_name \rangle$, $pso \langle service_name \rangle$ reaction $\langle reaction_name \rangle$ status</td>
</tr>
<tr>
<td>Fields Updated:</td>
<td>$pso$ state update, $pso$ state priority, $pso$ state details.</td>
</tr>
</tbody>
</table>

Table 7.22: The identity information for the System Status Agent

status fields and sends the workslip on.

**System Status Agent**

The System Status Agent tracks information sent to it by other agents in the $pso \langle service\_name \rangle$ status, $pso \langle service\_name \rangle$ error, $pso \langle service\_name \rangle$ reaction $\langle reaction\_name \rangle$ and $pso \langle service\_name \rangle$ reaction $\langle reaction\_name \rangle$ status fields. The agent must be configurable to allow the principal to track all of the services they are monitoring and the potential reactions that agents might be taking. The identification block for the System Status Agent is shown in Table 7.22.

When the System Status Agent receives an updated field in a workslip it incorporates the new information into its model of the system. For every service that the principal is monitoring it tracks information such as the status of the service, the length of time it has had that status and any of the reactions that agents may have taken in reaction to that service. It can then report on the status of the system by updating fields $pso$ state update, $pso$ state priority, and $pso$ state details with new information for the principal. The nature of the information that the status agent updates is configured by the principal to provide information in a useful fashion.
Principal Communication Agent

The Principal Communication Agent sends information to the principal in a way convenient for the principal to use. The principal must be able to configure the agents to look for particular information and to communicate it effectively to the principal. The identification block for the Principal Communication Agent is shown in Table 7.23.

The nature of the communication is dependent on the principal. Different PCAs can be configured to convey information in different ways. For example, the principal might want a text if the SSH service is down, but if there is an error that could be dealt with later then an e-mail might be the appropriate way.

The Principal Communication Agent does not provide an update that another agent can use in the system so it does not update any fields.

7.3.4 System Application

With the workslip fields identified and agents found which can read and update the fields appropriately, the principal can create a personal agent system to help them monitor important services by configuring the agents and running them.

As with the Aware Alarm Clock and Shopping List the agents are implemented generally as described above, with specific implementation details discussed below. The principal will need to configure each agent with the service it is monitoring or reacting to, as well as the
7.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

actual method for doing its work.

- The Service Testing Agents are implemented to test services in different ways. The basic version is to execute a script that can rely on functionality from the operating system to execute the test. For example the FTP service test attempts to download a file via FTP and the web server test attempts to download a file via HTTP.

- The Service Reaction Agents also execute scripts. These also take advantage of the operating system to attempt to restore the service.

- The System Status Agent is constructed to hold a map with a list of updates for each particular service. The principal can configure what changes in status should result in messages being sent.

- The Principal Communication Agent produces a text file of updates for the correct information at the correct priority. This is intended to allow for the testing of the system.

Again, as with the other two systems, a Dispatcher is used to coordinate the movement of workslips between agents. Each agent registers with the Dispatcher and provides a list of the workslip fields it will update and a list of the fields it reads. When the Dispatcher receives a new workslip with a field that the agent is interested in, it will give a copy of the workslip to that agent. This is also managed via the file passing system for the workslips.

The Proactive System Operator runs quite frequently, on the order of seconds or minutes depending on the services being monitored. A system regularly passes empty workslips to the Service Testing Agents, so that they monitor their systems, and send updates to the dispatcher, which then sends the updates on to interested agents.
Specific examples of the system in use are illustrated in Section 8.3, however it is also important to consider how the system for the Proactive System Operator compares to the Philosophical Guidelines for the Task and Workslip System as discussed in Chapter 5.

As discussed earlier, the task itself is well suited to the scheme, being regular repeated and informatic. Similarly the agents are also well suited. Each agent does exactly one task step and those task steps are relatively small. Each testing agent tests only one service one way and each reaction agent implements only one reaction. If a system needs several tests or several reactions then multiple agents can be used to provide the needed functionality. Agents are reusable with only a little reconfiguration, and should also be easily translated from one service to another depending on the nature of the service and the script the agent executes. Again the agent information blocks enforce clarity on requirements and production, with each agent reading and updating a small number of fields.

Again, TWPARI-J provides a unified mechanism to create and transport workslips so any agents developed using the framework automatically have the workslip easily shared between agents. The workslips are also capable of preserving history regarding the system. While one workslip may not hold the whole history of the system a compilation of workslips, such as what the System Status Agent compiles, provides a history that allows the state of the system over time to be seen. The fields of the workslip are also sufficiently specific, for each service there are fields for their information and for each field the potential values are either limited or supplementary to another field.

Finally the personal agent system is well suited to the task. First the task is personal, while it might be a task done as part of an organization and several principals might actually share agents, the configuration and organization of the agents must be for the principal’s benefit. The system is automatic in that agents are each able to operate independently and guided, particularly well in this case because the agents are able to communicate with
the principal about the work they are doing via the Principal Communication Agent. The system is both efficient and discreet, with discretion being managed by the PCAs sending only information to the principal in the specified ways in specified situations, the system is generally efficient as any test or reaction by the agents will only be a momentary task lasting a few seconds at most.

These three example systems illustrate the basic principles and concepts that the Task and Workslip Scheme for Personal Agents provides. They demonstrate the flexibility of the scheme to be used for different types of tasks and the demonstrate the flexibility of workslips as a communication and coordination media between simple agents. They show that while simple, the agents are able to accomplish tasks that are sufficiently complex to be of benefit to a principal in some aspect of their lives. The following chapter expands on these systems to consider how each might be used for different principals and to consider how a principal might benefit from using these systems.
Chapter 8

Evaluating TWPARI-J Systems

Evaluating personal agent systems effectively can be difficult. Most authors follow the approach of the Electric Elves in [CGK+01], where the authors deployed the system within their own research groups for a few weeks. This allowed them to see the effectiveness and potential problems with their systems, but provided at the end of the day a limited evaluation of the true effectiveness of the system

Other systems such as RADAR [GS07] have included more comprehensive user studies, exposing hundreds of novice users to the tasks their system was designed to solve, using different iterations of their system. This allowed them to see the advancement of their techniques as they worked to create and improve iterations of their system. This was an important project that took place over many years and involved the work of a large number of people, and allowed them to see that the later iterations of the system did assist people in the way that they were hoping.

When evaluating frameworks for creating personal agent systems, some authors [MS14]

\footnote{Often their research questions were much more focused on the outcomes of specific AI or machine learning techniques, rather than the technique as part of the personal agent system.}
have taken advantage of having classes of students available to implement personal agent systems. This has allowed them to compare the difficulty in creating new systems using their frameworks compared to other frameworks or other approaches to constructing software. However the evaluations were not able to show the usefulness of the systems after they were constructed.

To evaluate systems built using TWPARI-J, an ideal approach would be to merge all of these approaches. First select a group of principals and determine what tasks they have that are well-suited to the Task and Workslip Scheme. Then, assign several developers to create new agents using TWPARI-J, and have those developers work with the principals to create the personal agent systems they need. To demonstrate how multiple agents can fulfill the same task steps, have multiple developers create agents that use the scheme in different ways, so that the principals would have a selection of agents to choose from when designing. At the same time a second set of developers need to create standard software systems that address the same problems for the principals.

With this scenario it would then be possible to compare the standard software approach to the TWPARI-J approach and the Task and Workslip Scheme in general. Developers could evaluate the ease of constructing agents, and principals could evaluate the ease and flexibility of using agents to construct different personal agent systems. Principals could also evaluate different agents with regard to their usefulness and flexibility and a study could be done on common designs of personal agent systems, including agent usefulness. Given time it would also be possible to see how different principals implement similar tasks in different ways (and which are more effective) and to see how new agents can be developed to make constructing personal agent systems easier.

Unfortunately, constructing this type of comprehensive evaluation is not easily possible. The Task and Workslip Scheme is intended to be a way for principals to automate tasks
that are important to them, and so, providing a task for the principal to solve would not provide a true picture as to the schemes usefulness to that principal. Similarly, the creation of personal agent systems depends on the availability of many different agents to work on the task steps of the different principals’ tasks. The availability of many agents requires the availability of developers willing to create these agents (although, ideally evaluations would show that implementing agents using the Task and Workslips Scheme and TWPARI-J would be easier than alternative methods).

Without this wide-spread participation, it is difficult to evaluate the Task and Workslip Scheme and TWPARI-J in a real-world context, and within a single PhD thesis. Hopefully in the future it will be possible to construct these evaluations as these tools are adopted for use in a community. However, at this stage, an alternate approach should be taken when looking at the systems.

The first aspect of this evaluation was carried out in Chapter 7, where for each of the example tasks, the task and the implemented system were evaluated to see how well they fit the philosophical guidelines. It may be easy to say that the tasks fit the philosophical guidelines well, because they were the tasks chosen to illustrate the scheme, but hopefully the diversity of the tasks will help show that the scheme can be applied to a wide variety of problems.

The second aspect of this evaluation is to create different personal agent systems that fulfill the different tasks of different principals. In this case the principals will be artificial (much like Alice and Bob earlier), but will allow for an investigation of how different needs can be met for different principals in different ways. For each example system, this chapter will include several different “artificial” principals with real-world tasks and personal agent systems that can be used to fulfill those tasks. This will provide a view of the diversity of task instances that principals can have, and personal agent systems that can complete these
tasks. For each evaluation the system will obviously be able to complete the task and work relatively as efficiently as possible, which suggests that there will be no reason to compare the effectiveness and efficiency for each task instance. Instead the evaluation will focus on the organization and configuration of agents being applied to the principals’ task instance.

For each of the example tasks from Chapter 2, the following sections include a selection of task instances for principals and a description of a personal agent system to satisfy those tasks. This is followed by a discussion of the personal agent system and its effectiveness as an application of the implementation.

8.1 An Environmentally Aware Alarm Clock

For the Environmentally Aware Alarm Clock, a principal will want to change the time the alarm sounds with respect to some factor from the environment. While there are many, many different combinations of factors that a principal might consider important to their commute, there are basic elements regarding the personal agent system for a principal that can be evaluated. The following examples illustrate these basic elements and can stand-in for more complicated cases. Each example case is assigned an example principal, each of which has a specific instance of the alarm clock task for their system. Table 8.1 introduces the principals and their instance of the task.

Each case allows a demonstration of a particular aspect of the system:
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Case</th>
<th>Principal</th>
<th>Task Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>Owen</td>
<td>Lives in Calgary, Canada. Wants to move his commute forward by 20 minutes if frost is in the current conditions</td>
</tr>
<tr>
<td>$A_2$</td>
<td>Tala</td>
<td>Lives in Calgary, Canada. Wants to move her commute earlier by 10 minutes if the temperature is between $-10^\circ C$ and $-15^\circ C$, by 30 minutes if the temperature is between $-15^\circ C$ and $-20^\circ C$, and by 60 minutes if it is colder than $-20^\circ C$.</td>
</tr>
<tr>
<td>$A_3$</td>
<td>Thad</td>
<td>Lives in Calgary, Canada. Wants to move his commute 10 minutes earlier, or an hour earlier for any amount of snow less than 20cm (linearly). Additionally, wants to wake up 20 minutes earlier if there is frost in the current conditions.</td>
</tr>
<tr>
<td>$A_4$</td>
<td>Fiona</td>
<td>Lives in Vancouver, Canada. Wants to move her commute earlier by 10 minutes if there is less than 25mm of rain, if there is more than 25mm of rain, she wants to give up on her commute and have the alarm go 90 minutes later.</td>
</tr>
</tbody>
</table>

Table 8.1: Four artificial principals and the task instances that they would like to fulfill.

- In the first case, $A_1$, Owen wants to move his commute up by a set time if a given keyword is present in a given field (namely frost, in current conditions). This allows a demonstration of the basic behaviour of the keyword evaluator and the alarm control agent.

- In the second case, $A_2$, Tala, wants to use the value range evaluator to determine how cold it currently is and manually define steps for the change in alarm depending on that value. This allows a demonstration of the alarm clock reacting to different degrees of a value.

- In the third case, $A_3$, Thad wants to have the alarm linearly move the wake up time earlier based on the amount of snow. He also wants to have a keyword for frost added to the alarm update. This allows a test of the linear value evaluator and also the
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

combining of conditions in the Alarm Control Agent.

- In the fourth case, $A_4$, Fiona wants to have her alarm change step-wise based on the rain and also to move the alarm backwards if its not worth trying to commute. This allows a test of a different location for the weather source agent and also to show that the Alarm Control Agent can manage changing the alarm in different directions.

With the needs of the four artificial principals in mind, it is now possible to create systems that will work for these principals. The first step is to organize the agents and configure them to do what the principals need and the second step is to show how the agents behave.

8.1.1 Setting up the Personal Agent Systems

In Section 7.1, all of the agents and the basic arrangement of the alarm clock system were described. In this section the individual version of the task for each virtual principal will be paired with a particular organization of agents, which will include how each agent needs to be configured.

**Alarm Clock System $A_1$ for Owen**

Owen has the simplest requirements, in terms of agents needed, for his personal agent system. Owen will need one of each type of agent, a Weather Source Agent, a Weather Evaluation Agent, an Alarm Control Agent, and an Alarm Sounding Agent. These will be called $WSA$, $WEA$, $ACA$, and $ASA$ respectively. When workslips flow through the system, a workslip will start at the Weather Source Agent, then it will flow to the Weather Evaluation Agent, then to the Alarm Control Agent, and finally to the Alarm Sounding Agent. The flow diagram is shown in Figure 8.1.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.1: The workslip flow of Owen’s Environment Aware Alarm Clock, showing the flow of fields of interest from the Weather Source Agent (WSA), to the Weather Evaluation Agent (WEA), to the Alarm Control Agent (ACA), to the Alarm Sounding Agent (ASA).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>weathersourceagent.feedURL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td><a href="https://weather.gc.ca/rss/city/ab-52_e.xml">https://weather.gc.ca/rss/city/ab-52_e.xml</a></td>
</tr>
<tr>
<td>Purpose</td>
<td>The URL for the Environment Canada forecast for Calgary, Alberta</td>
</tr>
</tbody>
</table>

Table 8.2: Owen’s Configuration for the Weather Source Agent, WSA

Configuration for each agent is also relatively simple, each table below shows the important configuration for the agent (the full agent configuration information is included in Appendix B).

Owen’s WSA has the configuration shown in Table 8.2. In particular, it needs to know where to source its weather from. This is detailed in the configuration weathersourceagent.feedURL, which tells the agent which URL to look for to find an RSS feed from Environment Canada.

Owen configures WEA as shown in Table 8.3. In particular, he needs to configure its keyword evaluator. In addition to naming the evaluator, which will be used in the workslip field generated by the agent, the agent needs to know which field in the weather report it should look at, which keyword it should look for and what penalty should be applied if the keyword is present.²

²For clarification, the agent has a directory within its configurations that contains configurations for evaluators, where for each evaluator the configurations above are included in a separate file (or the range evaluator configurations). One agent can have several evaluators which will all be run on the same input workslip, and the result workslip fields will all be put in the same workslip to be send on. As it is generally closer to the philosophical guidelines to have one agent be responsible for one evaluation, usually there will
### 8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.name</th>
<th>Value</th>
<th>Frost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides a name for the evaluator, which is used internally and in the workslip field the agent produces</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.keyword.field</th>
<th>Value</th>
<th>weather.source.current.condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides the workslip field which the keyword evaluator should look at for the keyword.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.keyword.word</th>
<th>Value</th>
<th>frost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides the actual keyword that the principal would like the evaluator to look for</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.keyword.present_score</th>
<th>Value</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides the numeric score that the agent should report if the keyword (frost) is present in the workslip field (weather.source.current.condition).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Owen’s Configuration for the Weather Evaluation Agent, WEA

The evaluation is given a clear name, Frost, which is easy for Owen to recall as he will need to add the field this agent is producing to ACA. Owen also wants to look for frost currently occurring, which will be reported in the current conditions field using the keyword frost. Finally, Owen provides the score which the evaluator should return if frost is present in the current conditions. In this case Owen provides 100, although the actual value (in this circumstance) does not matter, only that the value is consistent between WEA and ACA.

Owen configures ACA as shown in Table 8.4. In particular, the Alarm Control Agent needs to track one or more alarms. Each alarm is loaded as part of the agent’s configuration and includes a name (as with the evaluation, used in the field that updates this alarm’s sounding time), a default time and the alarm modification table which tells the agent how only be one set of evaluator configurations per evaluation agent.
8.1. **AN ENVIRONMENTALLY AWARE ALARM CLOCK**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Morning</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the alarm, which is used internally and in the workslip field the agent produces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>06:00</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the time at which the alarm should sound if the system does nothing to alter it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.modification_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Field</td>
</tr>
<tr>
<td></td>
<td>evaluated.frost</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the logic for the agent to use when deciding when to change the time for the alarm. In this case if the field <code>evaluated.frost</code> has a score of at least 100, then the alarm should be changed to 20 minutes earlier.</td>
</tr>
</tbody>
</table>

Table 8.4: Owen’s Configuration for the Alarm Control Agent, ACA to adjust the alarm.

The alarm needs a name, in the same way the evaluator needed a name. This name will be used to label the field used by ASA to identify information about the alarm it is updating, and so Owen will need to be able to easily identify the alarm, in this case “Morning”. The default time is the time at which the alarm should sound if there is no frost in the current conditions. Each day ACA will reset the alarm to that time and then as the system runs, it will modify the time, if necessary. The alarm modification table provides the logic that the agent will use to determine if the time needs to be modified. In this case, the table is very straightforward, only including the one field the agent is interested in (`evaluated.frost`), the score at which the agent should act and the amount of time that the alarm time should be changed by (in minutes). Because the keyword evaluator only provides a single score, as long as the table is consistent with the value output by the evaluator then the alarm will
be modified if the keyword is observed. If the evaluation is from a value range evaluator then there will be different scores possible, and the time will only be changed if the score is sufficient to warrant it.

Owen configures ASA as shown in Table 8.5. As the alarm sounding agent only needs to know what time to sound the alarm (and possibly how to sound the alarm, but technical details such as that can be omitted here), the only configuration it needs is what workslip field it should be checking for. Here, Owen needs the alarm to check for its updated time in the \texttt{alarm.morning.time} field.

With these four agents configured, Owen’s personal agent system will be ready to run. When agents are first set up, ACA needs to receive an initial workslip to cause it to send the default alarm time to ASA. Then periodically WSA needs to be sent an empty workslip to cause it to begin working and to send weather information on to the rest of the system.

\textbf{Alarm Clock System} \textit{A}_2 \textbf{for Tala}

Tala has very similar requirements to Owen. She also needs, a Weather Source Agent, a Weather Evaluation Agent, an Alarm Control Agent, and an Alarm Sounding Agent. (\textit{WSA}, \textit{WEA}, \textit{ACA}, and \textit{ASA}). As with Owen, workslips will begin at the Weather Source Agent, then flow to the Weather Evaluation Agent, then to the Alarm Control Agent, and finally to the Alarm Sounding Agent. The flow diagram is shown in Figure 8.2. Tala uses the same agents in the same sequence, but she will provide (some) different configuration
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.2: The workslip flow of Tala’s Environment Aware Alarm Clock, showing the flow of fields of interest from the Weather Source Agent (WSA), to the Weather Evaluation Agent (WEA), to the Alarm Control Agent (ACA), to the Alarm Sounding Agent (ASA).

for the agents she uses.

The notable difference between Tala’s alarm clock and Owen’s will be in how WEA is configured, using the weather range evaluator rather than the keyword evaluator. The other configurations will be very similar. The tables in this section will show the important configurations (even the repeated ones), and the full configuration information for Tala’s Alarm clock is included in Appendix B.

Living in Calgary, Tala’s will configure WSA with the same configuration that Owen uses, the RSS feed address for the Environment Canada forecast, shown in Table 8.2.

Tala’s configuration for the WEA is shown in Table 8.6. Tala wants her system to react differently to different temperatures, so she needs to configure the WEA to use a value evaluator to look at the wind temperature field in the workslip, and then define a table which includes the scores she would like assigned the weather ranges.

Tala needs a name for the evaluator which will be used in the field produced by WEA, in this case, “Cold”. The field that the evaluator needs to look at, weather.source.current.temperature, contains the current temperature, which is stored as a float. Finally the evaluator needs to know how to evaluate the field. Tala provides the evaluator.value.table configuration, a table which allows the evaluator to assign a score, as determined by the principal, based on a low (inclusive) and high (exclusive) range. A value can be left blank if needed (the first line in the table allows for the lower than -20 to be accommodated).
Table 8.6: Tala’s Configuration of the Weather Evaluation Agent, WEA

Tala’s configuration for ACA, seen in Table 8.7, is again similar to Owen’s. The alarm must be named and given a default time. However Tala needs several extra entries in her alarm modification table to accommodate the different adjustments to her alarm time.

The Alarm Control Agent, ACA will consult the table and having several entries for the evaluated.cold field will apply the most suited one, which is the one with the highest threshold score that is passed. For example if the score was 175 (not a possibility in this configuration, but an example), then the clock would be set back by 30 minutes, if the score was 225, then the clock would be set back by 60 minutes.

The Alarm Sounding Agent, ASA, will be used and configured in a very similar way for all of the example principals. It really only needs to know the name of the alarm it is responsible for, all intelligence is contained in the Alarm Control Agent. As such, Tala’s configuration of the agent, seen in Table 8.8, only differs from Owen’s in the name of the alarm.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>alarm.name</td>
<td>Tala</td>
<td>Provides a name for the alarm, which is used internally and in the workslip field the agent produces</td>
</tr>
<tr>
<td>alarm.time</td>
<td>08:00</td>
<td>Provides the time at which the alarm should sound if the system does nothing to alter it.</td>
</tr>
<tr>
<td>alarm.modification_table</td>
<td></td>
<td>Provides the logic for the agent to use when deciding when to change the time for the alarm.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Score</th>
<th>Adjust</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluated.cold</td>
<td>200</td>
<td>-60</td>
</tr>
<tr>
<td>evaluated.cold</td>
<td>150</td>
<td>-30</td>
</tr>
<tr>
<td>evaluated.cold</td>
<td>100</td>
<td>-10</td>
</tr>
</tbody>
</table>

Table 8.7: Tala’s Configuration for the Alarm Control Agent, ACA

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.time_field</th>
<th>alarm.tala.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the time at which the alarm, Tala, needs to sound.</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.8: Configuration for Tala’s Alarm Sounding Agent, ASA

Alarm Clock System $A_3$ for Thad

Thad’s personal agent system will be different from Owen and Tala’s in two major ways. First, to identify the amount of snow, an agent is needed to further process the data available. Secondly, two different weather evaluation agents are needed one to handle the evaluation of snow and a second to handle the evaluation of frost.

Thad needs, a Weather Source Agent, an Alarm Control Agent, and an Alarm Sounding Agent ($WSA$, $ACA$, and $ASA$). Additionally, because the forecast is provided as a long text string, a special parsing agent will be needed to determine if there is snow in the
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.3: The workslip flow of Thad’s Environment Aware Alarm Clock, showing the flow of fields of interest from the Weather Source Agent (WSA), to the Forecast Parsing Agent (FPA), to the Weather Evaluation Agents (WEA\textsuperscript{a} and WEA\textsuperscript{b}), to the Alarm Control Agent (ACA), to the Alarm Sounding Agent (ASA).

forecast, and to put the amount of snow into the weather workslip in a useable form. That agent will be the Forecast Parsing Agent, which will be called FPA. He will need also two Weather Evaluation Agents, one for snow and one for frost, these will be WEA\textsuperscript{a} and WEA\textsuperscript{b} respectively, because they will require different configurations to operate.

Thad’s workslip flow will begin at the Weather Source Agent, then flow to the Forecast Parsing Agent, then to both Weather Evaluation Agents. The Alarm Control Agent will merge worksips that come from from the two WEAs, and the resulting workslip will flow finally to the Alarm Sounding Agent. The flow diagram is shown in Figure 8.3.

As Thad also lives in Calgary, his configuration for the WSA, is the same as Owen’s and Tala’s, seen in Table 8.2, the address for the Environment Canada Weather Report feed.

The Environment Canada Forecast is contained in one field in the weather report. The Forecast Parsing Agent needs to be able to find the correct field, determine which keywords it is looking for, and which regular expression is needed to extract the information. Thad’s configuration for FPA\textsubscript{3} is in Table 8.9.

The Forecast Parser Agent is more complicated to configure than most of the other agents
so far. It has a more specific and complicated task. The Environment Canada Forecast is broken up into days, with more specific information about the forecast on the nearer days and less specific information on later days. For example on a Friday during the day, the first entry will be the Friday entry, then one for Friday night, then one for Saturday, then one for Sunday. The days of the forecast are represented as a series of comma separated strings.

Thad must configure the FPA to find where to find the forecast to parse, in this case in the `weather.source.forecast` field. Then it needs to know which day of the forecast to check. Thad wants to know if there is snow in the immediate forecast, which would be the first entry in the forecast. Then the parser checks if snow is mentioned in the entry, and if it is, it will extract the number of centimetres from the entry using the regular expression.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.source.forecast</td>
</tr>
<tr>
<td>Purpose</td>
<td>The field the parser should attempt to extract information from.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.produce.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.parsed.snow</td>
</tr>
<tr>
<td>Purpose</td>
<td>The field the parser should produce if it has found helpful information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>first</td>
</tr>
<tr>
<td>Purpose</td>
<td>Which day of the forecast the parser should check for the keyword.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>snow</td>
</tr>
<tr>
<td>Purpose</td>
<td>The keyword the parser should look for.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.regex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>(.**) cm</td>
</tr>
<tr>
<td>Purpose</td>
<td>The regular expression the parser will use to try to capture the correct data from the forecast day, if it contains the keyword.</td>
</tr>
</tbody>
</table>

Table 8.9: Thad’s Configuration for Forecast Parser Agent, FPA
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Snow</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the evaluator, which is used internally and in the workslip field the agent produces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.keyword.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.parsed.snow</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the workslip field which the keyword evaluator should look at for the keyword.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.range.low_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0.0</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the minimum value that should be mapped</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.range.high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>20.0</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the maximum value that should be mapped</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.range.low_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the minimum score mapped to the minimum value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.range.high_score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>400</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the maximum score mapped to the maximum value</td>
</tr>
</tbody>
</table>

Table 8.10: Thad’s configuration for the instance of the Weather Evaluation Agent, \( WE A^a \)

Once this is finished it updates the new information to the new field \( \text{weather.parsed.snow} \), which will contain a float representation of the potential amount of snow in the forecast in centimetres (if there is any).

With the added field for snow, the Weather Evaluation Agents can be configured as seen in Table 8.10 and 8.11. Thad needs two instances of the Weather Evaluation agent, one to look at the snow and the other to look at frost. Each instance needs a different configuration, so are presented as \( a \) and \( b \).
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluator.name</td>
<td>Frost</td>
<td>Provides a name for the evaluator, which is used internally and in the workslip field the agent produces</td>
</tr>
<tr>
<td>evaluator.keyword.field</td>
<td>weather.source.current.condition</td>
<td>Provides the workslip field which the keyword evaluator should look at for the keyword.</td>
</tr>
<tr>
<td>evaluator.keyword.word</td>
<td>frost</td>
<td>Provides the actual keyword that the principal would like the evaluator to look for.</td>
</tr>
<tr>
<td>evaluator.keyword.present_score</td>
<td>100</td>
<td>Provides the numeric score that the agent should report if the keyword (frost) is present in the workslip field (weather.source.current.condition).</td>
</tr>
</tbody>
</table>

Table 8.11: Thad’s b Configuration for the instance of the Weather Evaluation Agent, WEA\textsuperscript{b}

For WEA\textsuperscript{a}, Thad needs to configure a range evaluator to provide a mapping of the snow to a score. The Range Evaluator takes two values and two scores. In this configuration, it has been given 0.0 (cm), the least amount of snow that can fall and 20.0 (cm) which is enough snow to make any further snow not excessively worse (in Thad’s mind). These are mapped to scores 0 and 400. The evaluator will linearly apply the mapping to determine the score, so a value of 10 cm of snow will return a score of 200, 5 cm a score of 100, and so on. Any value above 20 cm will be assigned a score of 400.

Thad’s configuration for WEA\textsuperscript{b} will be the same as the configuration Owen used for the WEA. It will look in the field weather.source.current.condition and see if frost is present at all. If so, the evaluator will add a field named evaluated.frost with a score of 100.

Since managing multiple evaluators is fairly straightforward for the Alarm Control Agent,
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Work</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the alarm, which is used internally and in the work slip field the agent produces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>07:00</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the time at which the alarm should sound if the system does nothing to alter it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.modification_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Score</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>50</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>100</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>150</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>200</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>250</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>300</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>350</td>
</tr>
<tr>
<td>evaluated.snow</td>
<td>400</td>
</tr>
<tr>
<td>evaluated.frost</td>
<td>100</td>
</tr>
</tbody>
</table>

Purpose: Provides the logic for the agent to use when deciding when to change the time for the alarm.

Table 8.12: Thad’s Configuration for the Alarm Control Agent, ACA

Each evaluator will return a score and the agent will evaluate those scores independently and apply the change in time either of them require. Thad’s configuration for the ACA is shown in Table 8.12.

The ACA does not do a linear interpretation on its own (although a replacement agent could be easily crafted that would implement this). Instead, Thad has configured the agent so that for every extra 50 score WEA \(^a\) provides the time is moved back 1/8th of an hour (7.5 minutes). Only the highest threshold surpassed for a field will alter the time. If the frost evaluation score is 100, ACA will apply that adjustment on top of any time change resulting from the snow. As such, if there is frost, but no snow then the time will be set to
6:40, if there is 2.5 cm of snow, then the time will be set to 6:33. If there is 25 cm of snow, then the time will be set to 5:40. If there is 2.5 cm of snow, but no frost then the time will be set to 6:53.

The Alarm Control Agent retains the recent information it has received, so if it receives one workslip with either snow or frost and then later a workslip for the other, it will continue to evaluate the two, as though they had arrived simultaneously.

Finally, the configuration of the Alarm Sounding Agent for Thad remains the same as for Owen and Tala. It only needs to know the field it should be looking for to set its time to sound. The configuration for ASA is found in Table 8.13.

### Alarm Clock System $A_4$ for Fiona

Fiona’s requirements are simpler than Thad’s, but slightly more complicated than Tala’s or Owen’s. She has only a single flow of workslips between agents, but needs to use a Forecast Parsing Agent to parse out the amount of forecasted rain. She needs a Weather Source Agent, a Forecast Parsing Agent, a Weather Evaluation Agent, an Alarm Control Agent, and an Alarm Sounding Agent ($WSA, FPA, WEA, ACA$, and $ASA$). Workslips in Fiona’s system will flow from $WSA$, to $FPA$, to $WEA$, to $ACA$, and finally to $ASA$, as seen in Figure 8.4.

First Fiona will need to configure $WSA$ differently to everyone else’s (although not significantly). Instead of the URL for the feed for Calgary’s weather information, she will need
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.4: The workflow of Fiona’s Environment Aware Alarm Clock, showing the flow of fields of interest from the Weather Source Agent (WSA), to the Forecast Parsing Agent (FPA), to the Weather Evaluation Agent (WEA), to the Alarm Control Agent (ACA), to the Alarm Sounding Agent (ASA).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>weathersourceagent.feedURL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td><a href="https://weather.gc.ca/rss/city/bc-74_e.xml">https://weather.gc.ca/rss/city/bc-74_e.xml</a></td>
</tr>
<tr>
<td>Purpose</td>
<td>The URL for the Environment Canada forecast for Vancouver, British Columbia</td>
</tr>
</tbody>
</table>

Table 8.14: Fiona’s Weather Configuration for Source Agent, WSA

the feed for Vancouver’s (as seen in Table 8.14).

To parse the rain from the forecast, Fiona will need to use the Forecast Parsing Agent in a similar way to what Thad did for snow. The configuration for FPA is shown in Table 8.15.

Fiona’s configuration for FPA is very similar to Thad’s, with only the produced field and keyword differing, with a substitution of rain for snow, and the regular expression changed parse values in millimetres rather than centimetres.

Fiona can configure the Weather Evaluation Agent WEA to look at the amount of rain and to adjust the score based on the amount, as seen in its configuration in Table 8.16.

This causes the evaluator to identify the two conditions Fiona was concerned with, rain less than 25 mm and rain greater than 25 mm. The agent assigns each a score that the ACA will be able to use to modify the time.

Fiona configures the ACA as seen in Table 8.17. In particular, she uses the alarm modification table to adjust the time backwards if the evaluator finds there is any rain or
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.source.forecast</td>
</tr>
<tr>
<td>Purpose</td>
<td>The field the parser should attempt to extract information from.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.produce.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.parsed.rain</td>
</tr>
<tr>
<td>Purpose</td>
<td>The field the parser should produce if it has found helpful information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>first</td>
</tr>
<tr>
<td>Purpose</td>
<td>Which day of the forecast the parser should check for the keyword.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>rain</td>
</tr>
<tr>
<td>Purpose</td>
<td>The keyword the parser should look for.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>parser.source.regex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>(.*) mm</td>
</tr>
<tr>
<td>Purpose</td>
<td>The regular expression the parser will use to try to capture the correct data from the forecast day, if it contains the keyword.</td>
</tr>
</tbody>
</table>

Table 8.15: Fiona’s Configuration for the Forecast Parser Agent, FPA

forwards if the rain is over a specific amount. This takes advantage of the fact that the highest threshold value for a given field is applied by the Alarm Control Agent.

Finally as with the other principals so far, Fiona’s configuration for the ASA only needs to provide which workslip field it needs to monitor to find its sounding time. The configuration for ASA is found in Table 8.18
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Rain</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the evaluator, which is used internally and in the workslip field the agent produces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.value.field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>weather.parsed.rain</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the workslip field which the keyword evaluator should look at for the keyword.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>evaluator.value.table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Purpose</td>
<td>The ranges and resultant scores the evaluator should return</td>
</tr>
</tbody>
</table>

Table 8.16: Fiona’s Configuration for the Weather Evaluation Agent, WEA

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Bike_Time</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the alarm, which is used internally and in the workslip field the agent produces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>08:30</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the time at which the alarm should sound if the system does nothing to alter it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>alarm.modification_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Field</td>
</tr>
<tr>
<td></td>
<td>evaluated.rain</td>
</tr>
<tr>
<td></td>
<td>evaluated.rain</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the logic for the agent to use when deciding when to change the time for the alarm.</td>
</tr>
</tbody>
</table>

Table 8.17: Configuration for Fiona’s Alarm Control Agent, ACA
### 8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>alarm.time_field</td>
<td>alarm.bike_time.time</td>
<td>Provides the time at which the alarm, Work, needs to sound.</td>
</tr>
</tbody>
</table>

Table 8.18: Configuration for Fiona’s Alarm Sounding Agent, ASA
8.1.2 The Personal Agent Systems in Action

When deployed for principals the agents will generally not be seen nor heard. They will run on the hosts they are assigned to and only agents at the “edges” of the system will interact with the principal if everything is going well. For the purposes of demonstrating the function of TWPARI-J, this section includes an “exploded view” of each agent running in a different terminal in a verbose mode.

Agents run in one of three different modes, Version, Registration and Run. In Version mode they print information about their operation, particularly which fields they read and update (an example of this is seen in Figure 8.5). This is intended to help set up and organize agents, and to make debugging agent behaviour simpler (the primary cause of errors in the system is mismatches in field names between agents, a topic discussed longer in the future work Section of Chapter 9. In Registration mode, they send their registration information to the dispatcher. In Run mode the agents loop while waiting for workslips to arrive.

Figure 8.5: A screen shot of Owen’s agents running in a exploded view on a MacBook Pro running macOS Sierra. The dispatcher in the top left corner is ready to run and all of the agents have printed their version information.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Once the agents are configured they can begin running. To run, the dispatcher is started first. As the agents are started they first run in Registration mode (see Figure 8.6), and then in Run mode (see Figure 8.7).

![Figure 8.6: A screen shot of Owen’s agents running in an exploded view on a MacBook Pro running macOS Sierra. The dispatcher is running and all of the agents have registered with the dispatcher to allow them to communicate.](image)

Once they are running, the agents act when they receive a workslip. For agents that react to the work of other agents, such as the ASA, or WEA this is a straightforward operation, they simply wait for an agent to produce a workslip with a field they are interested in. Other agents need to be prompted to work with an empty workslip. The primary example of this is the WEA which needs an empty workslip to prompt it to gather weather information and update the workslip to send to the other agents. The other example is the ACA which needs to receive one workslip to allow it to send the initial alarm information to the other agents in the system. An example of this is shown in Figure 8.8.

In order to manage this in a real system for a principal, an automated system would be needed to send empty workslips to agents at an appropriate time. An agent could be
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.7: A screen shot of Owen’s agents running in a exploded view on a MacBook Pro running macOS Sierra. Agents are now running, now the ACA and the WSA will be given blank workslips to begin processing.

developed to do this, however most operating systems provide a facility to manage operations based on time and it may be more reasonable to have an assistant script for the principal to prompt the agents.

Agents can run indefinitely, simply acting when prompted. To reset the alarm, the alarm system needs to be restarted once a day. While this may not seem like the simplest solution, it does allow the agents to be as simple as possible. More complex arrangements for handling alarms could also be added in future versions of the agent.
Figure 8.8: A screen shot of Owen’s agents running in an exploded view on a MacBook Pro running macOS Sierra. The WEA has just written a workslip that the dispatcher will read and then send on to the ACA.

The Weather

To demonstrate correct operation of the Alarm Clock System working correctly, weather information was recorded from Environment Canada for Calgary, Alberta (Area AB 52[Env18]) over several months from March 23, 2016 to August 4, 2016. This information was collected via RSS, using a modified version of the WSA. This agent stored the information in a format that was readable for other agents. To test the system without needing to rely on a change in the weather, a special version of the WSA, the Replaying Weather Source Agent, was developed. Instead of accessing the live weather report, it accesses stored weather information and creates a workslip using that information for the other agents.

The collected weather information is somewhat insufficient to demonstrate the exact criteria of the created principals used for evaluation. To manage this seven days of weather information from between March 23, 2016 and March 30, 2016 were selected, and modified slightly, to help demonstrate the effect of the principals concerns.
These weather instances are labeled $WC_1, ..., WC_7$ (for Weather, Calgary) and are used to demonstrate the behaviour of the systems for examples $A_1, ..., A_3$. Further weather has been constructed for $A_4, WV_1, ..., WV_7$. An example of the stored weather file for $WC_1$ is shown in Figure 8.9.

**System $A_1$ for Owen**

Owen wants his alarm clock to respond to frost in the current conditions. He wants to be able to get up 20 minutes earlier if there is frost. Weather instances $WC_1$ and $WC_2$ are used to demonstrate Owen’s situation. $WC_1$ does not contain frost and $WC_2$ does. This section shows two workslips produced by Owen’s system in these two weather instances. The first workslip is presented in full in Figures 8.10, 8.11, 8.12 and 8.13. Workslips from other weather instances are included in Appendix C.

In Figure 8.10 the two primary nodes of the workslip XML are displayed including the metadata, with information, such as the name, creation time and modification time. The second node is the workslip entries node which includes the fields and their values for the workslip.

The entries highlighted in red are produced by the Weather Source Agent, and include information about the weather, the order of the fields in the workslip is determined by JDOM 2.0 library and as such, are in no particular order for our purposes. Here, the current humidity was 34% and there were no warnings issued when then weather was recorded.

The unusual entry on this first page is the field `workslip.agents.processed` which is a special field included to help agents note when they have already processed a workslip, because all fields are left intact in the workslip as it is processed by agents, an agent could see a workslip it had already processed and processed it again without some mechanism to identify that it had already worked on it.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

In Figure 8.11, all of the entries are again produced by the Weather Source Agent. It is interesting to note the different types of information included in the workslip including the very long string of the forecast, but also the ISO formatted time stamps of the recorded and published times, and the simple floating point number for the temperature. It is also worth noting that as one of the earliest agents developed the WSA encodes all of these as strings, due primarily to lack of information about the type of information that could appear. With more surety now about the data, and better agent design techniques, they would be assigned types that are more restrictive to help other agents use the information.

In Figure 8.12, there are several more entries produced by the Weather Source Agent. The entry produced by the Weather Evaluation Agent is also included. In this case because frost is not present in the current condition field, the score produced by the WEA is 0.0. Figure 8.14 illustrates the outcome for the situation where frost is included in the forecast.

In Figure 8.13, there are still several more fields from the Weather Source Agent. It also includes the field from the Alarm Control Agent, which shows the alarm (morning) is set to go off at 6:00h. The Alarm Control Agent resets the alarm each day so the ACA and ASA are only communicating about the current day when communicating about the alarm. As with the update from the WEA, the field produced by the ACA can be seen in Figure 8.14.

These four figures show an ordinary workslip produced by an ordinary run of a Personal Agent system.

Figure 8.14, shows the workslip produced by a second run of Owen’s Weather Aware Alarm Clock, for which frost was added to the current condition. This figure shows only the relevant fields, those that changed and are important to the result of the work, other fields are committed for brevity.

The Frost can be seen in the field `weather.source.current.condition`, which corresponds to the evaluation of 150 in the field `weather.evaluated.frost.score`, and as a result the time
in `alarm.morning.setTime`, is set to 05:40h. Workslips in the following sections will also be edited to limited the amount of extraneous information.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.9: An example of the weather files used for the agents. In this case, example $WC_1$, recorded March 23, 2016.
Figure 8.10: The first page of Owen’s system’s workslip from $A_1$ with weather $W_1$. Entries highlighted in red are produced by the WSA. Entries that are not highlighted are meta entries.
Figure 8.11: The second page of Owen’s system’s workslip from $A_1$ with weather $W_1$. Entries highlighted in red are produced by the WSA.
Figure 8.12: The third page of Owen’s system’s workslip from $A_1$ with weather $W_1$. Entries highlighted in red are produced by the $WSA$, entries highlighted in blue are produced by the $WEA$. 
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.13: The fourth page of Owen’s system’s workslip from $A_1$ with weather $W_1$. Entries highlighted in red are produced by the WSA, entries highlighted in blue are produced by the ACA.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.14: Excerpts of Owen’s second workslip, from $A_1$ with weather $W_2$. The highlights show the differences from the workslip seem in $A_1$ with $W_1$. In particular the Frost in the current condition from the $WSA$, the evaluation of the $WEA$ and the time adjustment by the $ACA$. 

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workslip>
  <metaData>
    <name>DEFAULT_EMPTY_WORKSLIP</name>
    <creationTime>2018-09-15T17:23:42.641-06:00</creationTime>
    <modifiedTime>2018-09-15T17:23:43.589-06:00</modifiedTime>
  </metaData>
  <workslipEntries>
    <entry>
      <workslipField>
        <name>weather.source.current.temperature</name>
        <type>java.lang.String</type>
        <description>The current temperature</description>
      </workslipField>
      <workslipData>-0.6</workslipData>
    </entry>
    ...
    <entry>
      <workslipField>
        <name>weather.source.current.condition</name>
        <type>java.lang.String</type>
        <description>The current weather condition</description>
      </workslipField>
      <workslipData>Mist, Frost</workslipData>
    </entry>
    ...
    <entry>
      <workslipField>
        <name>weather.evaluated.frost.score</name>
        <type>java.lang.Float</type>
        <description>Evaluation of weather as evaluated by Frost Evaluator</description>
      </workslipField>
      <workslipData>150.0</workslipData>
    </entry>
    ...
    <entry>
      <workslipField>
        <name>alarm.morning.setTime</name>
        <type>org.joda.time.DateTimeField</type>
        <description>Alert interested agents that the time for alarm morning should now be the given time</description>
      </workslipField>
      <workslipData>1970-01-01T05:40:00.000-07:00</workslipData>
    </entry>
    ...
  </workslipEntries>
</workslip>
```
System $A_2$ for Tala

Tala wants her alarm clock to respond to cold temperatures in the current forecast. In particular she wants to be an hour early if the weather is colder than $-20^\circ C$, half an hour earlier if the weather is colder than $-15^\circ C$ and fifteen minutes earlier if the weather is colder than $-10^\circ C$. Weather condition $WC_1$ includes weather which is warm enough that her alarm should not be changed, $WC_2$ includes weather cold enough to move her alarm forward by 15 minutes, $WC_3$ includes weather cold enough to move her alarm forward by 30 minutes, and $WC_4$ includes weather cold enough to move her alarm forward by an hour. Workslips resulting from each of these conditions are included in Appendix C.

Figure 8.15 shows a workslip from Tala’s Alarm in the coldest possible weather. As before, the workslip is trimmed to only include the fields relevant for the alarm. These include the current temperature from the $WSA$ (in red), the evaluation by the $WEA$ (in blue) scoring the $-28.3^\circ C$ as 200, which is then translated by the $ACA$ (in yellow) to a wake up time one hour earlier than she would normally wake up.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.15: Excerpts of Tala’s fourth workslip, from $A_2$ with weather $WC_4$. The highlights show the differences from the Owen’s workslip, including the presence of the `weather.evaluated.cold.score` evaluation as produced the $WEA$ and the different alarm `alarm.tala.setTime` as produced by the $ACA$. 
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

System $A_3$ for Thad

Thad wants his alarm clock to respond to both snow in the forecast and frost in the current conditions. In particular, he wants his alarm clock to change further back based on the amount of snow from 0.0 cm to 20.0 cm, ranging up to an hour earlier if there is 20 cm of snow. If the current condition includes frost, then he wants the alarm to sound another twenty minutes earlier regardless of the snow. To show this the sample weather for Thad includes snow at 2 cm, 12 cm and 22 cm, both with frost and without.

Because the ACA for Thad works on the receipt of any weather evaluation, no one workslip will show the full state of logic behind the ACA acting. It will keep track of the last update it has seen for the fields it is monitoring and make its updates to the alarms accordingly. Figure 8.16, shows excerpts of the result of Thad’s system on W6, which shows that both his conditions for frost and the worst snow are met, and the alarm is moved an hour and 20 minutes early.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.16: Excerpts from Thad’s workslip from $A_3$ with weather $WC_6$. The alarm time can be seen to be set to 6:40am, because there is 20cm of snow forecast and frost in the current conditions. Only the evaluations from the snow are shown because this workslip was updated by the ACA.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

System $A_4$ for Fiona

Fiona wants her alarm to sound early if there is a little bit of rain and later if there is enough rain that she would rather work from home instead. Fiona also differs from the other principals in that her weather is based on the weather in Ottawa rather than the weather in Calgary. Her situation allows a demonstration that the Aware Alarm Clock systems are adaptable to different data sources and that adjustments to the clock can happen both earlier and later.

The weather for Fiona is adjusted from the forecast for Vancouver for the same time period as the weather examples for the, with three examples selected March 23, 24 and 25, and adjusted slightly to include an appropriate amount of rain to demonstrate her system in action. Weather $WV_1$ includes no rain, $WV_2$ includes 5mm of rain and weather $WV_3$ includes 26mm of rain.

The results of running her system are shown in Figures 8.17 and 8.18. In these two workslips the original forecast can be seen, as well as the work of the $FPA$ and the $WEA$. In Figure 8.17 the system can be seen making the small adjustment Fiona wanted to allow her to leave a little early in light rain and in Figure 8.18 the large adjustment to set the time later on the days when she will work from home.
8.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

Figure 8.17: Excerpts from Fiona’s system running on weather WV_2 where the parsing and evaluation can be seen, as well as the result of the ACA moving the alarm earlier by 10 minutes.
Figure 8.18: Excerpts from Fiona’s system running on weather WV₂ where the parsing and evaluation can be seen, as well as the result of the ACA moving the alarm later by 90 minutes.
8.2 An Efficient and Price Aware Shopping List

For the Shopping List seen in Chapter 7.2, different principals will want to have their shopping plans generated in different ways depending on their preferences for planning and their choice of stores. As with the Alarm Clock, there are many different combinations of factors that a principal may consider when selecting and configuring their agents for the shopping list. In general, for the Shopping List, the primary difference will be in selecting a shopping plan agent that plans according to their preferred style.

One difference between the Shopping List and the Alarm Clock is the availability of information. The Alarm clock is able to take advantage of forecast and historical weather information provided by the Government of Canada. Because information regarding grocery items provided for sale can be viewed as intellectual property and more to the point, this information frequently changes in provided format on the Internet, it is difficult to use real-world information when evaluating the Shopping List Systems.

To avoid either of the problems, the evaluations in this section will be done with three artificial stores, which will be used by each of the three artificial principals. These stores include Store A, which is a high end grocery retailer, Store B which is a discount grocery store, and Store C which is a smaller store local to the three principals. In this case, because there is limited benefit in illustrating differences in plan and store agents, the focus for these evaluations will be on the different planing agents the principals choose to use. Each principal will share store information agents and shopping lists.

There will be three example cases for the Shopping List evaluation and as with the Alarm Clock each will be exemplified with an artificial principal. These are listed in Table 8.19.

- In the first case, \( L_1 \), Olive wants to limit her travel time, so she will chose a Shopping Plan Agent which minimizes travel distance.
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Case</th>
<th>Principal</th>
<th>Task Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$</td>
<td>Olive</td>
<td>Wants the least amount of travel.</td>
</tr>
<tr>
<td>$L_2$</td>
<td>Taylor</td>
<td>Wants the least expensive plan possible.</td>
</tr>
<tr>
<td>$L_3$</td>
<td>Thea</td>
<td>Wants the plan that mostly closely matches their preferred items.</td>
</tr>
</tbody>
</table>

Table 8.19: Three artificial principals and the task instances of the Price Aware Shopping List that they would like to fulfill.

- In the second case, $L_2$, Taylor wants to ensure that his shopping plan is as inexpensive as possible, so he will choose a Shopping Plan Agent which minimizes overall cost.

- In the third case, $L_3$, Thea wants to ensure that her shopping plan matches her item preferences as much as possible, so she will choose a Shopping Plan Agent which maximizes item matching.

Rather than designing unique systems for each artificial principal, as for the Alarm Clock, in this case all three principals will share the same overall design, only switching the Shopping Plan Agent to illustrate that each of these principals needs only a minimal change to a common scheme to produce the results they would like.

### 8.2.1 Setting up the Personal Agent Systems

Each of the roles of the agents and their basic configurations in a personal agent system are discussed in Section 7.2.4. One primary difference is that the Store Information Agents have been replaced with a simplified version that takes a flatfile record of store information and uses it to populate the workslip field rather than to rely on the outside world, as discussed earlier. Otherwise their agents behave as originally discussed.

For each of their systems the principals will have three Store Information Agents for each of the three stores and one Shopping List Agent. These will also send information, via the
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

Figure 8.19: All three principals will have their information flow through the same organization of agents. Here information flows from the Shopping List Agent and each of the Store Information Agents to the Shopping Plan Agent which then constructs the best possible plan for the principal and sends it to the Principal Communication Agent to send on to the principal.

Each of the principals will configure the Shopping List Agent with the location of the file which holds the shopping list and the name of the field to share the list in. The relevant configurations are presented in Table 8.20. Each will keep their shopping lists in the same location for this example. Configuring the workslip field name allows the principal to ensure

---

3 The full configurations are presented in Appendix B
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shoppinglist.file.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>grocery_list.txt</td>
</tr>
<tr>
<td>Purpose</td>
<td>The location of the file where each principal will keep their shopping list.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shoppinglist.field_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>my_shopping_list.txt</td>
</tr>
<tr>
<td>Purpose</td>
<td>The name of the field which this agent will update with the shopping list information.</td>
</tr>
</tbody>
</table>

Table 8.20: The shared configuration for the Shopping List Agent, SLA

<table>
<thead>
<tr>
<th>Shopping List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 1% Milk</td>
</tr>
<tr>
<td>Smuckers Strawberry Jam</td>
</tr>
<tr>
<td>Olive Oil</td>
</tr>
<tr>
<td>Bananas</td>
</tr>
<tr>
<td>Oranges</td>
</tr>
</tbody>
</table>

Table 8.21: The shopping list shared by Olive, Taylor and Thea

that the Shopping List Agent and the Shopping Plan agent look in the same place to access the same information.

The example shopping list used for the evaluations here is shown in Table 8.21, contains a few items, enough to demonstrate the different shopping plans needed by each of the different agents. The first two items have both a brand and an item, the Alpha Milk and the Smuckers Strawberry Jam. The second two items, have no brand so the plan agent selecting by preference will not have a preferred brand to choose between each. The final item, Oranges, are not offered by any of the stores below, as this item was included to demonstrate how the system handles unobtainable items.

The principals will also share the same configurations for each of the three Store Information Agents. In normal deployment the agents would be configured with a name, a particular URL to retrieve the store information from, information for possibly a parser to correctly extract the information and some supplementary information to help inform the Shopping
Plan Agent. The Store Information Agents here need to be configured with a store name, a location for the source data, and a distance to the store. The configurations are presented in Tables 8.22, 8.23, and 8.24.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.info.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Store A</td>
</tr>
<tr>
<td>Purpose</td>
<td>The name of the Store with the associated information (used to generate field names).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.info.distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>15</td>
</tr>
<tr>
<td>Purpose</td>
<td>The principal’s average travel time to the store.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.datafile.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>store_a.data</td>
</tr>
<tr>
<td>Purpose</td>
<td>The location of file containing the store’s information.</td>
</tr>
</tbody>
</table>

Table 8.22: The shared configuration for the Store Information Agent A, SIA_A

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.info.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Store B</td>
</tr>
<tr>
<td>Purpose</td>
<td>The name of the Store with the associated information (used to generate field names).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.info.distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>45</td>
</tr>
<tr>
<td>Purpose</td>
<td>The principal’s average travel time to the store.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Store.datafile.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>store_b.data</td>
</tr>
<tr>
<td>Purpose</td>
<td>The location of file containing the store’s information.</td>
</tr>
</tbody>
</table>

Table 8.23: The shared configuration for the Store Information Agent B, SIA_B

For the evaluation here, each store represents a different purpose for the principals. Store A represents large chain grocery store. It has average to high prices, but a wider selection. Additionally, it is moderately close to the principals. Store B is a bulk discount store. It has
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>store.info.name</td>
<td>Store C</td>
<td>The name of the Store with the associated information (used to generate field names).</td>
</tr>
<tr>
<td>store.info.distance</td>
<td>1</td>
<td>The principal’s average travel time to the store.</td>
</tr>
<tr>
<td>store.datafile.name</td>
<td>store_c.data</td>
<td>The location of file containing the store’s information.</td>
</tr>
</tbody>
</table>

Table 8.24: The shared configuration for the Store Information Agent C, SIA\text{C}

<table>
<thead>
<tr>
<th>Item</th>
<th>Brand</th>
<th>PPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1% Milk</td>
<td>Alpha</td>
<td>2.00</td>
</tr>
<tr>
<td>1% Milk</td>
<td>Palm</td>
<td>1.99</td>
</tr>
<tr>
<td>Strawberry Jam</td>
<td>Smuckers</td>
<td>3.00</td>
</tr>
<tr>
<td>Strawberry Jam</td>
<td>Kraft</td>
<td>1.50</td>
</tr>
<tr>
<td>Olive Oil</td>
<td>Oliveri</td>
<td>3.48</td>
</tr>
<tr>
<td>Olive Oil</td>
<td>Store A</td>
<td>2.4</td>
</tr>
<tr>
<td>Bananas</td>
<td>Dole</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 8.25: The shared store information for Store A, including the Item, Brand and price per unit (PPU)

low prices and a moderate selection of items. It is much farther away from the principals than the other two stores. Finally, Store C is close to the principals, but has high prices and a very limited selection. Tables 8.25, 8.26, and 8.27, show the stock of each store.
With common SLA and SIA agents, the Shopping Plan Agent needs to be configured differently for each principal. As does the Principal Communication Agent. The configurations for each principal with their preferred organization are presented in the following sections.

**Shopping List System \( L_1 \) for Olive**

Olive’s preference is to minimize the travel distance in each plan the system generates for her. As such, she selects a Distance Minimizing Shopping Plan Agent, which minimizes the total distance in the plan it generates. This Shopping Plan Agent shares most of its configuration with the other variants, which the other two principals will use. In general, it needs to know what stores it should look for store information from in the workslip and where in the workslip the principal’s shopping list will be shared. The configurations for her

<table>
<thead>
<tr>
<th>Store B Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Item</strong></td>
</tr>
<tr>
<td>1% Milk</td>
</tr>
<tr>
<td>Strawberry Jam</td>
</tr>
<tr>
<td>Olive Oil</td>
</tr>
<tr>
<td>Olive Oil</td>
</tr>
<tr>
<td>Bananas</td>
</tr>
</tbody>
</table>

Table 8.26: The shared store information for Store B, including the Item, Brand and price per unit (PPU)

<table>
<thead>
<tr>
<th>Shopping List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Item</strong></td>
</tr>
<tr>
<td>1% Milk</td>
</tr>
<tr>
<td>Strawberry Jam</td>
</tr>
<tr>
<td>Olive Oil</td>
</tr>
<tr>
<td>Olive Oil</td>
</tr>
<tr>
<td>Bananas</td>
</tr>
</tbody>
</table>

Table 8.27: The shared store information for Store C, including the Item, Brand and price per unit (PPU)
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>store.names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Store A, Store B, Store C</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the names of the three stores that will be used when producing Olive’s shopping plan.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shopping.list.field_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>my_shopping_list.txt</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the name of the field which the agent will find the principal’s shopping list in.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shopping.plan.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>olive</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the shopping plan. Used to identify the plan and produce workslip field names.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>plan.distance.penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>.2</td>
</tr>
<tr>
<td>Purpose</td>
<td>Parameter to tune exactly how much distance traveled should be penalized in Olive’s shopping plan. Penalty is applied per minute travel</td>
</tr>
</tbody>
</table>

Table 8.28: Configuration for Olive’s Distance Minimizing Shopping Plan Agent.

Olive’s Shopping Plan Agent needs to know which stores it should check for store information. In this case, it is configured with the names of all three stores in the scenario. It also needs to know about Olive’s shopping list, which is stored in the field my_shopping_list.txt as set in the configuration for the Shopping List Agent. Olive’s plan needs a name so that it can be shared with other agents, and so the name olive is used which will produce the field shopping_plan.olive. Finally, the agent needs to know what penalty Olive assigns to distance traveled. The weighting used applies a planning penalty to the cost of items at a further distance. In this case, if the agent is considering a can of tuna which costs $1.00 at Store A and $0.75 at Store B will consider the Store B can at $4.50 (which is the $0.75 cost times the 30 km extra travel, adjusted by the penalty value of 0.2).
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>shopping.plan.name</td>
<td>olive</td>
<td>Provides a name for the shopping plan. Used to identify the plan and produce workslip field names.</td>
</tr>
<tr>
<td>communication.file.location</td>
<td>olive_shopping_plan.txt</td>
<td>Provides a location where the agent is able to write a file for the principal to read.</td>
</tr>
</tbody>
</table>

Table 8.29: Configuration for Olive’s Principal Communication Agent.

Olive’s Principal Communication Agent is largely the same as the other two principals’ in this scenario, however it will need to be configured with the name of her plan so that it knows what fields to read when looking for shopping plan information. The Principal Communication Agent is very simple in its construction, because as with the Alarm Sounding Agent for the Aware Alarm Clock, a complex agent is not useful for evaluation. This agent records a file with information for the principal. The location of that file is also configured for the agent. Olive’s configuration for the Principal Communication Agent is shown in Table 8.30.

**Shopping List System $L_2$ for Taylor**

Taylor’s preference is to minimize the total cost in the shopping plans the system generates for him. He selects a Cost Minimizing Shopping Plan Agent, which produces a plan with the lowest total over all cost for all of the items on Taylor’s shopping list. This agent needs slightly fewer configurations than Olive’s Shopping Plan Agent, and most of the configurations are very similar. The configurations for Taylor’s Shopping Plan Agent are shown in Table 8.30

As with Olive’s agent, Taylor’s Shopping Plan Agent needs to know which stores to check
for store information. Again the agent is configured with the names of all three stores in the scenario. Similarly the agent needs to know which workslip field will contain Taylor’s shopping list and again this will be the field `my_shopping_list.txt`. Finally the agent needs to know the name of the plan, which is used to identify the plan and to produce name of the workslip field where the plan will be updated.

Taylor’s Principal Communication Agent will be largely the same as Olive’s, with the most important configuration being the name of Taylor’s shopping plan, and additionally the name of the file where Taylor’s communication will be written. The Configuration for Taylor’s Principal Communication Agent is shown in Table 8.31
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shopping.plan.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>taylor</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the shopping plan. Used to identify the plan and produce workslip field names.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>communication.file.location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>taylor_shopping_plan.txt</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a location where the agent is able to write a file for the principal to read.</td>
</tr>
</tbody>
</table>

Table 8.31: Configuration for Taylor’s Principal Communication Agent.

**Shopping List System $L_3$ for Thea**

Thea’s preference is to ensure that her preferred items are included in her plan over alternatives that might be nearer or cheaper. As such, she selects an Brand Prioritizing Shopping Plan Agent. The configuration for this agent is very similar to the other Shopping Plan Agent configurations in this scenario. The agent will need to know where to find the store information, where to find Thea’s shopping list, what the name of the Thea’s shopping plan is, and, in her case, what penalty should be applied when selecting an item which is not her preferred brand. The configurations for her Shopping Plan Agent are shown in Table 8.32.
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<table>
<thead>
<tr>
<th>Configuration</th>
<th>store.names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Store A, Store B, Store C</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the names of the three stores that will be used when producing Olive’s shopping plan.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shopping.list.field_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>my_shopping_list.txt</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides the name of the field which the agent will find the principal’s shopping list in.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>shopping.plan.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>thea</td>
</tr>
<tr>
<td>Purpose</td>
<td>Provides a name for the shopping plan. Used to identify the plan and produce workslip field names.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>plan.mismatch.penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>4</td>
</tr>
<tr>
<td>Purpose</td>
<td>Parameter to tune exactly how much the agent should dislike an item mismatch.</td>
</tr>
</tbody>
</table>

Table 8.32: Configuration for Thea’s Brand Prioritizing Shopping Plan Agent.

As with the other two agents, Thea’s Shopping Plan Agent needs to know which stores to look for information from, that will be all three stores in the scenario, and the agent needs to know where to look for Thea’s shopping list which will again be the file my_shopping_list.txt. The shopping plan needs a name for identification and to generate workslip field names, which in this case will be thea. Finally the Item Prioritizing Shopping Plan Agent needs to know how strongly Thea would like her preferred brand prioritized over other brands. In this case with a penalty of 1.75 Thea’s preferred brand of tuna Brand A, might have a can at $1.00, and Thea would only consider a can of Brand B if it was $0.25 or less or 4 times less expensive.

Thea’s Principal Communication Agent is largely the same as the other two. The agent will be configured with the name of Thea’s shopping plan and file where it will communication
Configuration | shopping.plan.name  
---|---
Value | thea  
Purpose | Provides a name for the shopping plan. Used to identify the plan and produce workslip field names.

Table 8.33: Configuration for Thea’s Principal Communication Agent.

with Thea. The configuration for Thea’s Principal Communication Agent is shown in Table 8.33.

### 8.2.2 The Personal Agent Systems in Action

The system for executing agents for the Aware Shopping List is the same as the one for the Aware Alarm Clock. Agents each run from their own script and can run in version, registration or continuous execution (run) mode. Agents communicate via passing workslips as files to the dispatcher which determines which agents will be interested in the workslip and passing the files onto to the next interested agent.

Unlike the Aware Alarm Clock, all of the evaluations of the system share the same setup and agents, a Shopping List Agent, three Store Information Agents for Stores A, B, and C, a Shopping Plan Agent and a Principal Communication Agent. Workslips pass from the Shopping List Agent and the three Store Information Agents to the Shopping Plan Agent and on to the Principal Communication Agent (as shown in Figure 8.19).

For each of the following evaluations the only difference will be the plan generated by the Shopping Plan Agent. With Olive representing case $L_1$ looking for the least amount of overall travel (baring unusual cost), Taylor representing case $L_2$ looking for the least expensive...
shopping trip, and Thea representing case $L_3$ looking for the plan that best matches her preferred items and brands. To show the system in action the Olive’s evaluation will include four workslips sent from the Shopping Plan Agent, showing how the plan evolves as more information is received from each shop. The others will show only the last workslip sent by the Shopping Plan Agent.

**System $L_1$ for Olive**

The evolution of the plan produced by the Shopping Plan Agent for Olive is shown over the workslips below. In the first, shown in Figure 8.20, we can see that because the Shopping Plan Agent does not have any information about the stores it notes that for all of the items in the shopping list, there is no store available to buy them from.

In the second workslip, shown in Figure 8.21, the first store information arrives. Store A has a wide selection of items, and so is able to provide all of Olive’s (except Oranges). Since it is the only store the Shopping Plan Agent agent is aware of, it is the closest and so all items are assigned to that store in the plan.
Figure 8.20: Excerpts from Olive’s workslip from $L_1$. The shopping list from Olive’s Shopping List Agent can be seen (highlighted in blue) in field `shopping_list.mylist` and the initial plan from the Shopping Plan Agent (highlighted in red) can be seen in field `shopping_plan.distance`.

In the third workslip, shown in Figure 8.22, information from Store B arrives. In this case because Store B is significantly further away compared to Store A. The plan does not change when it is updated. According to Olive’s preferences it is better still to shop for all her items at Store A.

In the fourth workslip, shown in Figure 8.23, information from Store C arrives. Because Store C is closer than Store A, the plan is updated so that most of the items on the list are moved to be picked up at Store C. However, milk at Store C is so expensive that even after the distance penalty is applied to the cost of Milk at Store A, the cost at Store A
remains cheaper. As such Olive’s Plan continues to have getting milk slated for Store A. This demonstrates that the principal can adjust the system to trade off cost and distance depending on their values. If Olive had wanted shorter travel time even at the expense of $12 a Litre milk, then she could raise the penalty for distance higher and have the system adapt accordingly.
Figure 8.21: Excerpts from Olive’s workslip from $L_1$. The plan from the Shopping Plan Agent (highlighted in red) can be seen in field `shopping.plan.distance`. The information from Store A can be seen highlighted in Blue, including the store information and the distance to the store.
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

Figure 8.22: Excerpts from Olive’s workslip from $L_1$. The plan from the Shopping Plan Agent (highlighted in red) can be seen in field `shopping.plan.distance`. The information from Store B can be seen highlighted in Blue, including the store information and the distance to the store. Notice that the plan does not change between this and Figure 8.22.
Figure 8.23: Excerpts from Olive’s workslip from $L_1$. The plan from the Shopping Plan Agent (highlighted in red) can be seen in field `shopping_plan.distance`. The information from Store C can be seen highlighted in Blue. Notice that as Store C is much closer, the SPA chooses it as the store for most items in the plan, except for milk for which Store C charges an extremely high amount, so milk remains with Store A.
System $L_2$ for Taylor

Taylor would like his shopping to be as inexpensive as possible. As such his Shopping Plan Agent is configured to find the items which are the cheapest possible, even if the brand is not what Taylor might prefer.

The operation of the system is the same as it was for Olive, the Shopping Plan Agent first sees an update from the Shopping List Agent, then from the Store Information Agent for Store A, then Store B, then Store C. Each time it outputs a plan. In this case when the update from Store B is seen most of the items are moved from Store A to Store B because they are less expensive. The only exception to this is the Jam which remains cheaper at Store A. When the update from Store C arrives, the information is evaluated by the Shopping Plan Agent, but the plan is not updated, because nothing is less expensive at Store C. This result can be see in Figure 8.24
Figure 8.24: Excerpts from Taylor’s workslip from $L_2$. The plan from the Shopping Plan Agent (highlighted in red) can be seen in field \texttt{shopping.plan.distance}. The plan calls for most items to be purchased at Store B, except for milk which remains less expensive at Store A.
8.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

System $L_3$ for Thea

Thea would like her shopping plan to allow her to purchase her preferred brand of items. As such her Shopping Plan Agent is configured to consult brand as a priority and then price as a secondary concern. The process for this system in action was the same as for the other two, with the Shopping Plan Agent receiving an update first from the Shopping List Agent, then from the Store Information Agent for Store A, then Store B, then Store C. In each case it updates the plan if necessary.

The two items in the shared shopping list with brands are the jam and the milk. In both cases, the Shopping Plan Agent schedules those items to be purchased at the store where they can be found and for the best price. The items without brand preference are scheduled according to the best price. Store C does not offer preferred brands at better prices and so is not chosen when creating the plan.
8.3 An Informative and Proactive System Operator

For the Proactive System Operator, a principal will want to react (or have the system react) to different situations that can arise within their technology services. As with the other example systems there are many different situations which may arise and many different organizations of services that the principal may be monitoring.

As with the Alarm Clock and the Shopping List, there are some aspects regarding the system which can be shown with exemplars and evaluated. In this case, there are two of...
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Case</th>
<th>Principal</th>
<th>Task Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>Oscar</td>
<td>Wants to receive an immediate alert for some of the systems he works with and an lower priority alert for some of the other systems.</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Tress</td>
<td>Wants to receive an immediate alert for some of the systems she works with, and for the PSO to automatically react to them.</td>
</tr>
</tbody>
</table>

Table 8.34: Two artificial principals demonstrating different use cases of the Proactive System Operator.

those situations, one where a principal only wishes to be alerted to a situation ($P_1$), and one where a principal wishes the PSO to attempt to fix the problem, as well as to be be alerted ($P_2$). These two situations are introduced with artificial principals in Table 8.34.

With the needs of these two artificial principals, Oscar, looking for monitoring, and Tress, looking for reaction, it is now possibly to create systems they can use. As with the other two systems, the first step will be to organize the necessary agents and configure them. Once complete it will be possible to organize the agents into personal agent systems, and then to see how the personal agent systems respond to the needs of the principals while reacting to different situations.

8.3.1 Setting up the Personal Agent Systems

The agents and basic organization of a Proactive System Operator are outlined in Section 7.3. In this section personal agent systems will be customized for the different needs of Oscar and Tress. They will share some agents, in order to allow the systems to react to the same situations, but they will also have some different agent configurations and some different agents as well.

Both principals will have the same set of Service Testing Agents, one for a Web Server and one for an SSH server. Then each will have a different arrangement of the other agents.
The agent that is used for the evaluation is a simplified version, which tests for the services if a specific file is available and that it has the correct status in it. This simplification replaces more complex agents that use other methods such as attempting to download a know file from a service or attempting to log into a service. The configuration for the Simplified Service Testing Agent is shown in Tables 8.35 and 8.36.

These two agents will produce workslip fields regarding the status of these systems, which the other agents the principals choose will use to inform their own work.
Proactive System Operator $P_1$ for Oscar

Oscar requires a simpler system than Tress, with only the shared Service Testing Agents (STA), the System Status Agent (SSA), and the Principal Communication Agent (PCA). Workslips flow from the two Service Testing Agents ($STA_{SSH}$ and $STA_{HTTP}$), to the System Status Agent, and onto four Principal Communication Agents, which each represent a different channel of communication between the Agents and Oscar. $PCA_{Text}$ sends a text message to his phone, $PCA_{Email}$ sends him an e-mail, $PCA_{Ticket}$ enters a ticket in his issue tracking system, and $PCA_{Log}$ simply logs the information. The flow for Oscar’s system is shown in Figure 8.26.
Figure 8.26: The workflow flow of Oscar’s Proactive System Operator, showing the flow of work slips from the two Service Testing Agents ($\text{STA}_{\text{SSH}}$ and $\text{STA}_{\text{HTTP}}$), to the System Status Agent (SSA), and on to the four Principal Communication Agents (PCAs).
Oscar configures his System Status Agent to reflect that he would like an immediate notification of the HTTP service being down, and a lower priority notification of the SSH service being down. He configures the Agent with the services he would like it to monitor, the situations he would like reports about and the priority with which he would like to be notified. The important configurations for his SSA are shown in Table 8.37.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Configuration</th>
<th>service.status_table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HTTP,</strong></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
</tr>
<tr>
<td><strong>SSH,</strong></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Situation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP → DOWN</td>
<td>ULTRA</td>
</tr>
<tr>
<td>UP → ERROR</td>
<td>HIGH</td>
</tr>
<tr>
<td>ERROR → DOWN</td>
<td>LOW</td>
</tr>
<tr>
<td>DOWN → ERROR</td>
<td>LOW</td>
</tr>
<tr>
<td>DOWN → UP</td>
<td>REGULAR</td>
</tr>
<tr>
<td>ERROR → UP</td>
<td>REGULAR</td>
</tr>
</tbody>
</table>

Table 8.37: Oscar’s configuration for his System Status Agent.

Oscar’s configuration shows his level of concern with the two services. He needs to know immediately if the HTTP service has gone down and almost immediately if it has had an error or if it has started working again. He needs to know about the SSH service going down or experiencing an error, but less immediately than the HTTP service and if the SSH service is already not working and changes how it does not work, he is even less interested in knowing.

Oscar configures four Principal Communication Agents, depending on the immediacy with which he needs to know about a situation. His highest priority method of communication is a text message sent to his phone, sent by the agent \( PCA_{text} \). His next highest priority method of communication is an e-mail, sent by the agent \( PCA_{email} \). His next highest priority
Configuration | state.priority  
---|---  
Value | ultra  
Purpose | The priority level this agent looks for in workslips with information about the state of the system.

| Configuration | state.log.file.name  
---|---  
Value | ultra.log  
Purpose | The file name of the log file for this agent.

Table 8.38: The configuration for Oscar’s Ultra Priority Logging PCA

| Configuration | state.priority  
---|---  
Value | high  
Purpose | The priority level this agent looks for in workslips with information about the state of the system.

| Configuration | state.log.file.name  
---|---  
Value | high.log  
Purpose | The file name of the log file for this agent.

Table 8.39: The configuration for Oscar’s High Priority Logging PCA

method of communication is a ticket in his issue tracking system, submitted by the agent \( PCA_{ticket} \). His lowest priority method of communication is to have the information placed in a log, by the agent \( PCA_{log} \).

For purposes of demonstration in this evaluation, however, there will be four \( PCAs \) for Oscar, but each will store its information in a log which will be shown when looking at the system in action below. The configuration for these four agents is shown in Tables 8.38, 8.39, 8.40 and 8.41.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>regular</td>
</tr>
<tr>
<td>Purpose</td>
<td>The priority level this agent looks for in workslips with information about the state of the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.log.file.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>regular.log</td>
</tr>
<tr>
<td>Purpose</td>
<td>The file name of the log file for this agent.</td>
</tr>
</tbody>
</table>

Table 8.40: The configuration for Oscar’s Regular Priority Logging PCA

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>low</td>
</tr>
<tr>
<td>Purpose</td>
<td>The priority level this agent looks for in workslips with information about the state of the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.log.file.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>low.log</td>
</tr>
<tr>
<td>Purpose</td>
<td>The file name of the log file for this agent.</td>
</tr>
</tbody>
</table>

Table 8.41: The configuration for Oscar’s Low Priority Logging PCA

With the agents configured this way, Oscar’s Proactive System Operator is ready to work. The Service Testing Agents need to be provided with an empty workslip to begin working and their first workslips will include information about the state of the services that the System State Agent will need to use to build its model of the current state of the system. Periodically thereafter the agents need to be provided with workslips again to be prompted to test their services.

**Proactive System Operator $P_2$ for Tress**

Tress’s system will be more complex than Oscar’s because she will use the Service Reaction Agents in addition to the shared Service Testing Agents, and her own System State Agent and Principal Communication Agents. The flow of workslips through her system will be similar
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

Figure 8.27: The workflow flow of Tress’s Proactive System Operator, showing the flow workslips from the two Service Testing Agents ($ST_{\text{SSH}}$ and $ST_{\text{HTTP}}$), to the two Service Reaction Agents ($SRA_{\text{SSH}}$ and $SRA_{\text{HTTP}}$) and to the System Status Agent (SSA), and then on to the three Principal Communication Agents (PCAs).

To Oscar’s. Workslips will begin at the Service Testing Agents ($ST_{\text{SSH}}$ and $ST_{\text{HTTP}}$), and will then flow to the Service Reaction Agents ($SRA_{\text{SSH}}$ and $SRA_{\text{HTTP}}$), as well as the System State Agent (SSA) before going to her Principal Communication Agents, $PCA_{\text{Text}}$, $PCA_{\text{Email}}$, $PCA_{\text{Ticket}}$, and $PCA_{\log}$ which all function in the same ways as Oscar’s. The flow for Tress’s system is shown in Figure 8.27.

Tress configures each of her Service Reaction Agents to react to the systems being down. In the case of the HTTP service, she could configure the reaction agent to post a message to an intranet page, or some other resource so that users of the system could see the system
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Configuration</th>
<th>service.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>HTTP</td>
</tr>
<tr>
<td>Purpose</td>
<td>The service this agent is checking for status updates from.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>service.status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>DOWN</td>
</tr>
<tr>
<td>Purpose</td>
<td>The condition the service named in service.name which this agent will react to.</td>
</tr>
</tbody>
</table>

Table 8.42: The configuration for Tress’s $SRA_{HTTP}$.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>service.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>SSH</td>
</tr>
<tr>
<td>Purpose</td>
<td>The service this agent is checking for status updates from.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>service.status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>DOWN</td>
</tr>
<tr>
<td>Purpose</td>
<td>The condition the service named in service.name which this agent will react to.</td>
</tr>
</tbody>
</table>

Table 8.43: The configuration for Tress’s $SRA_{SSH}$.

status before she even begins to react. Alternatively in the case of the SSH service, she could configure the agent to call a script to restart the server providing the SSH service. For the purposes of the evaluation shared here, both are configured to simply note in a log that an reaction to the service occurred. The configurations for her two Service Reaction Agents are shown in Tables 8.42, and 8.43.

Tress’s System State Agent also requires more configuration than Oscar’s because it must be able to handle information it receives from the Service Reaction Agents as well as from the System Testing Agents. Additionally Tress handles situations in a different way than Oscar, so has different priorities for different arising situations. Tress’s configuration for her SSA is shown in Table 8.44.

Tress is primarily concerned with services going from working to not, and has less of
a priority for dealing with services changing between non-working statuses or the services being restored. She also finds that regardless of the system there is not a situation where she needs to immediately fix a problem so the highest priority she uses is high. For both systems she wants to know if the reaction attempt failed at a medium priority, but if they succeeded she only finds that to be a low priority.

As with Oscar’s system, Tress’s PCAs could be configured to react to her in a number of different ways, such as text, message, or e-mail. For the purposes of this evaluation, her three PCAs will each store information for Tress in a log of the correct priority information.
The configurations for her PCAs are shown in Tables 8.45, 8.46 and 8.47.

With her agents organized and configured, Tress’s Proactive System Operator can now be run. As with Oscar’s system the Service Testing Agents need to be provided with an empty workslip to begin working, and they will update that workslip with the status of the system they are working on and send that on for the other agents to work with. Periodically thereafter, the Service Testing Agents agents will be prompted again with empty workslips which will allow them to test their services.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>low</td>
</tr>
<tr>
<td>Purpose</td>
<td>The priority level this agent looks for in workslips with information about the state of the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>state.log.file.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>low.log</td>
</tr>
<tr>
<td>Purpose</td>
<td>The file name of the log file for this agent.</td>
</tr>
</tbody>
</table>

Table 8.47: The configuration for Tress’s Low Priority Logging PCA

8.3.2 The Personal Agent Systems in Action

As with the Aware Alarm Clock and the Aware Shopping List, the Proactive System Operator uses a simple shared environment for all agents. The agents share a file space which they use to communicate via file passing and a dispatcher routes workslips to the interested agents. Agents are each executed from scripts in one of the three execution modes, version, registration or continuous execution (run).

For the Proactive System Operator agents run continuously with updates being sent from the System Test Agents on to either System Reaction Agents or System Status Agents. As these agents finish work they send their updates on to other agents, eventually ending with the Principal Communication Agents which update the principal. Examples of this can be seen if Figures 8.26 and 8.27. These systems produce more workslips than the others because they are continuously reacting to changes in the system environment. As such the examples here only include a sample of the output to demonstrate how the workslips are updated by the System State Agent and how the system reacts to particular situations.

In the first example for Oscar, the “early” part of the system only includes System Test Agents and the System State Agent which allows a demonstration of the system state agent detecting a service going down and then back up again. In the second example for Tress, the
“early” part of the system includes both System Test Agents and System Reaction Agents. The System State Agent handles the information from the System Test Agents the same way that Oscar’s will, these are omitted an only the workslip updates based on information from the System Reaction Agents is shown.

**System $P_1$ for Oscar**

In this scenario, Oscar’s SSH service briefly experiences an error. The SSH Service Test Agent at three points in time sends an update that the service is up, then in error, and then is up again. When the SSH Service Test Agent first sends an update with the status of error, the System State Agent notes the change and updates its internal evaluation of the state of the system (moving the SSH service from “UP” to “DROPPED”). This update is seen in Figure 8.28 where the updates from the SSA are added to the workslip it received from the $STA_{SSH}$. Oscar rated the SSH service going from up to error as a high priority event, so the priority of the update is set to high and the details of the situation that SSH has experienced an error 2 is included in the detail field. The earlier updates from the $STA_{SSH}$ can also be seen in the Figure 8.28 where the status of the SSH service is ERROR and the error detail is 2.

Later the SSH service stops experiencing the error and returns to the ‘UP’ state again. The $STA_{SSH}$ notices this and accordingly changes the state of the service in its internal model to ‘RESTORED’. Oscar’s configuration also rates this as a high priority event, so when the updates are put in the workslip, the priority is entered as ‘HIGH’ again. The updates can be seen in Figure 8.29 including the new state of the SSH service and the priority. Note that there is no detail information because the system is not able to provide any extra details (historical context would need to be gathered from older versions of the workslip). Again the update from the $STA_{SSH}$ remain in the workslip, in this case, only a
status update, since there is no error condition.

If the SSH service instead had simply stopped working, the updates would have been similar, although instead of the detail including information about the error the detail would simply be that the service was not available. Similarly if the HTTP service had experienced an error or stopped working, the updates would be similar, but reflect the difference in service.
Figure 8.28: A workslip from Oscar’s PSO, at this point in time the SSH service has gone from UP to ERROR, which the SSA evaluates as the service dropping. The updates from the STA can be seen in blue, and the updates from the SSA can be seen in red.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

Figure 8.29: A later workslip from Oscar’s PSO, at this point in time the SSH service has gone from ERROR back to UP, which the SSA evaluates as the service being restored. The updates from the STA can be seen in blue, and the updates from the SSA can be seen in red.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

System $P_2$ for Tress

Tress’s system is running in the same situation as Oscar’s, with her SSH service experiencing an error at one point and then later working correctly again. Because of the similarity of their system configurations, Tress will receive the same updates as Oscar did, from the $STA_{SSH}$ and the $SSA$. Additionally Tress’s system contains Service Reaction Agents which are responsible for detecting a change in a service (as received from a Service Testing Agent) and reacting to it, in this case by sending an alert to an outside source.

Figure 8.30 contains a workslip as it has been updated by the $SSA$ after receiving the update from the $SRA_{SSH}$. In this case the workslip with the update from the $SRA_{SSH}$, would be received after the update from the $STA_{SSH}$ (which both agents are interested in, and so both receive a copy from the dispatcher when it is sent out by the Service Testing Agent). This can be seen in that the status of the system has been changed from “DROPPED” to ‘DOWN’.

The update from the $SRA_{SSH}$ is also included, in this case indicating that the alert operation succeeded. If for some reason it had not succeeded, then the update would be that the attempt had failed. In either case, the $SSA$ takes the information about the success and the detail provided by the $SRA_{SSH}$ and gathers it the detail field to provide supplemental information for Tress.
Figure 8.30: A workslip from Tress’s PSO, at this point in time the SSH service has gone from UP to ERROR, and the SRA has reacted which the SSA now notes. The updates from the SRA are in yellow, the updates from the SSA are in red and the updates from the STA SSH are in blue.
8.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

The three different personal agent systems built using the Task and Workslip Scheme show how a common set of Agents can be organized in different ways to provide different benefits for different principals. In each case with the correct organization of agents and configuration, the principal can customize the system to meet their own specific needs.

In each case, the systems and the agents are designed following the philosophical guidelines in Chapter 5 and this shows that if the guidelines are followed then the principles of personal agents systems can succeed, giving principals control over their computational systems, but still getting benefit from flexibly personalized software.

Of the three systems, the Aware Alarm Clock and the Proactive System Operator are most closely designed in accordance with the guidelines, keeping the agents as small as possible for each sub-task they do. In these cases the system has relatively simple information stored in each workslip field and this increases the relative flexibility which allows adding new agents to the system which can do new sub-tasks.

The Aware Shopping List differs slightly in that the agents are larger, doing more sub-tasks in each agent than the other two systems. This results in more complicated information being sent between agents and increases the complexity of creating a new agent to work with the system. In this case having a designer who was new to developing systems using the Task and Workslip Scheme shows how using the scheme to develop software is somewhat counter intuitive. However even in this case the scheme still allows flexibility and allows developers to create new agents relatively simply. Similarly, principals can combine and configure these agents to produce an intelligent and helpful shopping list.

With these evaluations of the Task and Workslip Scheme, the reference implementation, and the systems built using them, it is possible to draw some conclusions regarding this approach to building personal agent systems and to look ahead to the work that can yet be done in this area.
Chapter 9

Conclusion and Future Work

A personal agent system allows a principal to construct a system out of agents to work together to solve their problems. The Task and Workslip Scheme for Personal Agent Systems provides a structured yet flexible way for agents to cooperate to work on these tasks. The scheme provides flexibility by allowing principals to combine the agents that meet their specific needs by having those agents share the structured workslips. The structure of the workslips ensures that each agent is able to easily communicate about what information it can provide to other agents and what information it needs from other agents.

The novel contributions in The Task and Workslip Scheme are several fold. Primarily the definition of a workslip, a structured system of data areas, passed between agents at each step of a task is novel. The definitions of principal, task, agent, environment and personal agent systems provide a unique approach to agent construction and organization within the area of personal agent systems. These concepts are refined by the philosophical guidelines, which provide a focus for developers creating agents and for principals creating personal agent systems.

The Task and Workslip Scheme is flexible in that it allows different groups of people to
assemble different agents to work on their tasks. Broadly, any task which is regular, repeated and informatic is well suited to a personal agent system developed using the scheme. The three systems demonstrated here show that the same core concepts of the scheme fit well into three different tasks, monitoring the principal’s alarm-time in response to the environment, monitoring the principal’s preferred stores for the best prices for the principal’s shopping list, and proactively monitoring the principal’s network of services to keep them up to date and react before they can if possible.

Within these different tasks however, different systems could be designed to fulfill the wishes of different principals. For the Aware Alarm Clock, four different principals were able to use the system in different ways combining different agents to the effect of adjusting their alarm time in the desired way. For the Aware Shopping List, three different principals were able to use the system in different ways to find different plans for their shopping trips (in this case even just changing one agent significantly allows for different principals to receive different results). For the Proactive System Operator, two different principals were able to design different systems to react to changes in their network environments.

In each of these examples the common concepts and guidelines from the scheme are shown in action. Agents are focused on one task step and are written to be as simple as possible within the given constraints of the system. Agents are reusable between principals doing similar tasks and agents are defined via their transformation of workslip fields they read into the fields they can update. Workslips provide a structure for the agents to share information intelligently, allowing them to communicate about general and specific information with enough structure that there is no confusion about what information each field in the workslip contains. Workslips also allow agents (and people observing them) to see what information has been provided before, which helps in the process of understanding why agents made certain decisions.
All of the personal agent systems also demonstrate another benefit of the Task and Workslip Scheme, that it allows the principals using the systems to keep their personal information to themselves. The systems all work within the environments the principals already have and do not require sharing information with third parties. None of the agents are so complex that they would task the principals’ computing resources. Additionally each of the systems are able to communicate with the principals in configurable ways.

**Future Directions**

The Task and Workslip Scheme has many areas in which it can grow and many more research questions which have yet to be answered. Some of these are practical, and relate to usage of the scheme to develop tools that people can commonly use, some are more academic focusing on the underlaying principles and concepts in Multi-Agent Systems, and other fields of Computer Science as well.

The first step of the possible future work involves creating ecosystems of principals, agents and agent developers. These ecosystems are necessary to fulfill the concept of many different principals working on their own tasks with many different shared agent, and will allow significant further research because they will allow study of the function and use of the scheme solving actual problems provided by actual principals. This will allow much richer studies of the effectiveness of the scheme, as well as provide insights into the niche in technology where the scheme fits.

To support the creation of these ecosystems, research is required on several fronts. One is the integration of the scheme into concepts such as agent-stores, which would be analogous to app stores for current operating systems, this includes matters of authentication to ensure that an agent is what it says it is, security and verification to ensure that an agent does what
it says it does, and only that, and social factors such as ensuring that agent developers are appropriately and fairly credited and compensated for their work.

Additionally, work should continue expanding and refining the tools to construct agents and agent systems. This includes improvements to TWPARI-J including implementing more protocols for agents to share worksips and simplifying elements of the agent interface to make developing new agents easier. Another reference implementation in another language, such as Python, would also assist developers. At a minimum an interface that allows agents to be written in other languages would help expand the reach of the scheme. Finally infrastructure tools that allow principals to visually construct personal agent systems, and find agents that are able to provide needed fields would also be beneficial.

Another area of interesting research is in embedding the agents within the physical environment of the principal. This idea is already allowed for within the concepts of the scheme, but is not investigated in depth. Having agents that are able to observe and react to the physical environment around the principal provides different factors to be investigated, including restrictions on the size and capabilities of agents, increased complexity of the role of the agent cell, different methods of agent communication and social factors such as how principals use and organize their personal agent systems when those systems are obviously spread out around them.

Within the scheme itself there are still interesting concepts to be investigated. These include using the scheme to support more complex interactions such as auctions or elections, not to change the organization of the system, but to allow agents to adjust their reactions to complicated situations. Additionally systems to handle conflict between the agents is an interesting area of study.

Finally, the focus of the Task and Worksip Scheme has been personal agent systems supporting a single principal. This has allowed the work on the scheme to avoid certain
pitfalls that have been problematic for other systems and approaches. However the fact remains that many tasks people do are in concert with other people. Finding effective methods for principals to work together using personal agent systems is also an important area.
Index

Agency, 9
Agent, 67
  Assignment, 64
  Instance, 69
  Operation, 69
  Read Fields, 68
  Updated Fields, 68
  Work, 69
Agents Set, 63
Alice and Bob, 11, 78
ANTICO, 42
Arnor, 44
Aware Alarm Clock, 11, 140
  Agents, 148
  Application, 159
  Evaluation, 197
    In Action, 217
    Set Up, 199
  Requirements, 141
  Workslip, 142
Aware Shopping List, 15, 165
  Agents, 170
  Application, 175
  Evaluation, 237
    In Action, 249
    Set Up, 238
  Requirements, 166
  Workslip, 167
BDI-Architecture, 47
CALO, 34
COLLAGEN, 28
Communication Structure, 61
Communication Structure Initial Value, 62
Communication Structure Update Time, 62
Communication Structure Value, 61
Computational Resources, 63
Electric Elves, 28
ELIZA, 24
Environment, 62

278
Environmental State, 62, 63
FIPA, 51
Goal Set, 71

History
   Artificial Intelligence, 24
Hive, 52
Hosts, 63

Knowledge Navigator, 26
Master Workslip, 57, 60
Microsoft Office Assistant, 25
MYCIN, 25
Mycroft, 26

Natural Language Processing, 24
Personal Agent System, 71, 105
Personal Agent Systems, 28
Personal Assistant, 26
Potential Workslip Set, 60
Principal, 9, 56
Proactive System Operator, 19, 179
   Agents, 185
   Application, 190
   Evaluation, 259
   In Action, 272
   Set Up, 260
   Requirements, 181
   Workslip, 181

RADAR, 38
Task, 56
Task and Workslip Scheme, 1, 55
   Philosophical Guidelines, 73
   Automatic Personal Agent Systems, 97
   Efficient Personal Agent Systems, 99
   History Bearing Workslip, 90
   Informatic Tasks, 84
   Personal Personal Agent Systems, 95
   Regular Tasks, 83
   Repeated Tasks, 80
   Reusable Agents, 87
   Shareable Workslip, 89
   Simple Agents, 85
   Specific Workslip, 92
   Well Defined Agents, 88
   Task Instance, 57
   Task Step, 56, 57
   Trace, 72
   Tropos, 44
TWPARI-J, 102

Agent, 105, 115
Agent Cell, 105, 116, 120
Agent Configuration, 117
Agent Core, 105, 116
Agent Execution Handler, 116, 122
Agent Implementation, 123
Agent Information, 116
Agent Interface, 116
Communication, 126
Development, 130

Development, 133

Execution, 134
Planning, 130
Dispatcher, 103, 127
Infrastructure, 125
Master Workslip, 115
Personal Agent Systems, 137
Script, 129

Shareable Workslip Format, 113

Workslip, 106

Implementation, 110
Interface, 109
Workslip Field, 112
Workslip Lifecycle, 114

Update, 72

Watson, 26

Wolfram Alpha, 26

Workslip, 60

Workslip Field, 58, 59

Workslip Field Information, 59

Workslip Field Value, 59

World Information, 63, 65

World Value, 66

XCON, 24

Xipho, 44
Bibliography


the deck" attack on software updates: Solution by distributed recommendation
of testers. In *2013 IEEE/WIC/ACM International Joint Conferences on
Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 2,
pages 293–300, Nov 2013.

[Bar16] Mark Barley. Search Automation for Deals Using the Task and Workshop
Scheme. Technical report, University of Calgary, Department of Computer
Science, 12 2016.

[BPC+06] Pauline Berry, Bart Peintner, Ken Conley, Melinda Gervasio, Tomás Uribe,
and Neil Yorke-Smith. Deploying a personalized time management agent.
In *Proceedings of the Fifth International Joint Conference on Autonomous
Agents and Multiagent Systems*, AAMAS ’06, pages 1564–1571, New York,
NY, USA, 2006. ACM.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John

[CGK+01] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean
Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves:
Applying agent technology to support human organizations. In *Proceedings of
the Thirteenth Conference on Innovative Applications of Artificial Intelligence

[CGK+02] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean
Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves:


[SBC+06] Aaron Steinfeld, Rachael Bennett, Kyle Cunningham, Matt Lahut, Pablo-Alejandro Quinones, Django Wexler, Dan Siewiorek, Paul Cohen, Julie


Appendix A

TWPARI-J: Java Documentation

This Appendix includes the system documentation for TWPARI-J. The table below provides a system overview of all components and the sections that follow include documentation for each Java package.

Class Hierarchy

Classes

- java.lang.Object
  - personalagents.AgentInformation (in A.1.3, page 301)
  - personalagents.AgentInformationSerializer (in A.1.4, page 307)
  - personalagents.AgentResources (in A.1.5, page 309)
  - personalagents.agentutilities.FloatExtractor (in A.2.1, page 319)
  - personalagents.cell.AgentCell (in A.3.2, page 322)
  - personalagents.cell.ReflectiveAgentClassFactory (in A.3.3, page 326)
  - personalagents.cell.executionmodel.ExecutionHandlerFactory (in A.4.6, page 337)
• personalagents.cell.executionmodel.FileWritingExecutionHandler (in A.4.7, page 339)
  • personalagents.cell.executionmodel.ContinuousFilePassingExecutionHandler
    (in A.4.4, page 332)
  • personalagents.cell.executionmodel.EmptyWorkslipOutputExecutionHandler
    (in A.4.5, page 334)
  • personalagents.cell.executionmodel.OneRunFilePassingExecutionHandler (in A.4.9, page 343)
    • personalagents.cell.executionmodel.FileWritingRegistrationHandler (in A.4.8, page 342)
    • personalagents.cell.executionmodel.RegistrationExecutionHandler (in A.4.10, page 346)
    • personalagents.cell.executionmodel.VersionExecutionHandler (in A.4.11, page 347)
    • personalagents.composableAgent.ComposableAgent (in A.5.3, page 350)
    • personalagents.configuration.ConfigurationSet (in A.6.1, page 351)
    • personalagents.dispatcher.DispatchTesting (in A.7.11, page 373)
    • personalagents.dispatcher.DispatcherCell (in A.7.9, page 371)
    • personalagents.dispatcher.DispatcherTestRunner (in A.7.10, page 372)
    • personalagents.dispatcher.FileReadingRegistrationHandler (in A.7.12, page 374)
    • personalagents.dispatcher.FileReadingWorkslipIntakeHandler (in A.7.13, page 376)
    • personalagents.dispatcher.FileWritingWorkslipOutputHandler (in A.7.14, page 378)
    • personalagents.dispatcher.FullFileWorkslipArchive (in A.7.15, page 380)
    • personalagents.dispatcher.LightTextWorkslipArchive (in A.7.16, page 381)
    • personalagents.dispatcher.SilentAlive (in A.7.20, page 387)
    • personalagents.dispatcher.TickAlive (in A.7.21, page 389)
Class Hierarchy

- personalagents.masterworkslip.MasterWorkslip (in A.8.1, page 396)
- personalagents.workslip.DefaultObjectDataSerializer (in A.10.3, page 411)
- personalagents.workslip.Workslip (in A.10.4, page 413)
- personalagents.workslip.WorkslipField (in A.10.5, page 422)
- personalagents.workslip.WorkslipSerializer (in A.10.6, page 427)

Interfaces

- java.lang.Runnable
- personalagents.dispatcher.RegistrationHandler (in A.7.1, page 360)
- personalagents.dispatcher.WorkslipIntakeHandler (in A.7.5, page 366)
- personalagents.Agent (in A.1.1, page 297)
- personalagents.AgentCore (in A.1.2, page 301)
- personalagents.cell.AgentClassFactory (in A.3.1, page 321)
- personalagents.cell.executionmodel.AgentExecutionHandler (in A.4.1, page 328)
- personalagents.cell.executionmodel.AgentExecutionHandlerFactory (in A.4.2, page 330)
- personalagents.cell.executionmodel.RegistrationHandler (in A.4.3, page 331)
- personalagents.composableAgent.InputHandler (in A.5.1, page 349)
- personalagents.composableAgent.Processor (in A.5.2, page 349)
A.1. PACKAGE PERSONALAGENTS

• personalagents.dispatcher.WorkslipArchive (in A.7.4, page 364)
• personalagents.dispatcher.WorkslipIntakeHandlerFactory (in A.7.6, page 367)
• personalagents.dispatcher.WorkslipOutputHandler (in A.7.7, page 368)
• personalagents.dispatcher.WorkslipOutputHandlerFactory (in A.7.8, page 369)
• personalagents.workslip.ObjectDataSerializer (in A.10.1, page 406)
• personalagents.workslip.WorkslipInterface (in A.10.2, page 408)

Exceptions

• java.lang.Object
• java.lang.Throwable
• java.lang.Exception
• java.lang.RuntimeException
• personalagents.configuration.MissingConfigurationException (in A.6.2, page 357)

A.1 Package personalagents

Package Contents

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>297</td>
</tr>
<tr>
<td>AgentCore</td>
<td>301</td>
</tr>
</tbody>
</table>

Agent ................................................................. 297
The core working component of an agent system, which transforms a workslip from input to output states.

AgentCore ............................................................ 301
Does the core work of an agent.
A.1. PACKAGE PERSONALAGENTS

Classes

AgentInformation ........................................................................................................... 301

The information to describe an agent.

AgentInformationSerializer ......................................................................................... 307

Class to write AgentInformation as an XML Document and construct

Agent Information from an XML Document

0.2.0 - Moved getFieldNameElement to WorkslipSerialzer.serializeWorkslipField and inflateWorkslipField to WorkslipSerialzer.inflateWorkslipField

0.2.1 - Added contact string (see AgentInformation (in A.1.3, page 301))

AgentResources ............................................................................................................. 309

Provides common resources that may be needed thought an agent.

A.1.1 Interface Agent

The core working component of an agent system, which transforms a workslip from input to output states. It is usually composed within an AgentCell (in A.3.2, page 322) which manages retrieving workslip information from the outside world. Each agent is executed in a core loop which calls the takeInput method, to take in a Workslip (in A.10.4, page 413), then the work to transform the workslip and then produceOutput to produce a workslip to be sent on. The other methods allow the cell or other calling entity to describe the agent. Generally the constructor for each Agent will need to establish the fields the agent will handle and load configuration for the agent to run properly.

Declaration

public interface Agent
All known subinterfaces

ComposableAgent (in A.5.3, page 350)

All classes known to implement interface

ComposableAgent (in A.5.3, page 350)

Method summary

getAgentInformation() Returns the AgentInformation (in A.1.3, page 301) which describes this agent.

getDefaultConfiguration() Returns the default ConfigurationSet (in A.6.1, page 351) for this Agent.

getMasterWorkslip() Returns the MasterWorkslip (in A.8.1, page 396) for this Agent.

produceOutput() Returns the updated Workslip (in A.10.4, page 413).

takeInput(Workslip) Takes in a Workslip (in A.10.4, page 413) for the Agent to work on.

work() Transforms the currently taken Workslip (in A.10.4, page 413).

Methods

• getAgentInformation

AgentInformation getAgentInformation()
Returns the AgentInformation (in A.1.3, page 301) which describes this agent.

- **Returns** – agent information describing this agent.

- **getDefaultValue**

  `configuration.ConfigurationSet getDefaultValue()`

  - **Description**
    
    Returns the default ConfigurationSet (in A.6.1, page 351) for this Agent. This provides a configuration that is loaded before any other configuration. This allows the developer to provide a base-line configuration for any critical parts of the Agent.

    - **Returns** – default configuration for this Agent.

- **getMasterWorkslip**

  `masterworkslip.MasterWorkslip getMasterWorkslip()`

  - **Description**
    
    Returns the MasterWorkslip (in A.8.1, page 396) for this Agent. This master list includes all of the fields that might be read or updated by this agent.

    - **Returns** – the master workslip for this agent.

- **produceOutput**
A.1. PACKAGE PERSONALAGENTS

workslip.Workslip produceOutput()

- **Description**
  Returns the updated Workslip (in A.10.4, page 413). After the workslip has been worked on this method allows it to be returned to be sent on to other agents.

- **Returns** – the updated workslip

• **takeInput**

void takeInput(workslip.Workslip workslip)

- **Description**
  Takes in a Workslip (in A.10.4, page 413) for the Agent to work on. The loaded workslip will then be transformed when the work method is called.

- **Parameters**
  
  * workslip – the workslip to be transformed.

• **work**

void work()

- **Description**
  Transforms the currently taken Workslip (in A.10.4, page 413). This should take the values currently loaded into the workslip and the Agent’s own state and use it to produce a meaningful update to the workslip.
A.1.2  Interface AgentCore

Does the core work of an agent. They are dynamically loaded and able to process a particular situation (as extracted from a Workslip (in A.10.4, page 413)).

Declaration

```java
public interface AgentCore
```

Method summary

- `process()` Do the work needed for an Agent (in A.1.1, page 297) to transform a Workslip (in A.10.4, page 413).

Methods

- `process`

  ```java
  void process()
  ```

  - Description

    Do the work needed for an Agent (in A.1.1, page 297) to transform a Workslip (in A.10.4, page 413).

A.1.3  Class AgentInformation

The information to describe an agent. Update 0.2.0: Added agent contact string to allow agents to determine where (and how) they should be communicated with.
Declaration

public class AgentInformation
    extends java.lang.Object

Constructor summary

AgentInformation(String, String, String, String, String, String, String, List, List)
Constructs a new description of an agent.

Method summary

equals(Object)
getAgentName() Returns the name of the agent.
geAuthor() Returns the name of the organization or person who wrote the agent.
geContact() Returns the contact information for the Agent.
geDescription() Returns a text description of the agent.
geFieldsRead() Returns a list of the fields this agent may read to do its work.
geFieldsWritten() Returns a list of the fields this agent may update as it works.
geIssuer() Returns the name of the organization or person who provides this agent.
geVersion() Returns the version of the agent.
hashCode()
toString()
Constructors

• AgentInformation


    - Description

        Constructs a new description of an agent.

    - Parameters

        * agentName – the unique name of the agent
        * issuer – the name of the issuer of the agent
        * author – the name of the developer of the agent
        * version – the updated version of the agent
        * contact – contact information for the issuer or the author
        * description – a brief textual description of what the agent does
        * fieldsRead – a list of the fields the agent will read to work
        * fieldsWritten – a list of the fields the agent might update when it works

Methods

• equals
public boolean equals(java.lang.Object arg0)

• getAgentName

public java.lang.String getAgentName()

– Description

Returns the name of the agent. The name should be unique across issuers.

– Returns – the name of the agent.

• getAuthor

public java.lang.String getAuthor()

– Description

Returns the name of the organization or person who wrote the agent.

– Returns – the name of the agent author.

• getContact

public java.lang.String getContact()

– Description

Returns the contact information for the Agent. This information should reflect
the current maintainer or the provider.
A.1. PACKAGE PERSONALAGENTS

- **Returns** – the contact information for the maintainer or provider.

- **getDefinition**

  ```java
  public java.lang.String getDescription()
  ```

  - **Description**
    Returns a text description of the agent. This should provide a description that a principal can use to determine if this agent does the sub-task they are looking to have done.

  - **Returns** – a text description of the agent.

- **getFieldsRead**

  ```java
  public java.util.List getFieldsRead()
  ```

  - **Description**
    Returns a list of the fields this agent may read to do its work.

  - **Returns** – a list of fields the agent reads.

- **getFieldsWritten**

  ```java
  public java.util.List getFieldsWritten()
  ```

  - **Description**
    Returns a list of the fields this agent may update as it works.
- **Returns** – a list of the first this agent updates.

- **getIssuer**

  ```java
class PersonalAgents {
    public java.lang.String getIssuer() {
      // implementation
    }
  }
  ```

  - **Description**
    Returns the name of the organization or person who provides this agent.

  - **Returns** – the name of the agent issuer.

- **getVersion**

  ```java
class PersonalAgents {
    public java.lang.String getVersion() {
      // implementation
    }
  }
  ```

  - **Description**
    Returns the version of the agent. This should be updated each time the agent is recompiled. The version information and history should be provided by the agent issuer separately.

  - **Returns** – the current version of the agent

- **hashCode**

  ```java
class PersonalAgents {
    public native int hashCode() {
      // implementation
    }
  }
  ```

- **toString**
A.1.4 Class AgentInformationSerializer

Class to write AgentInformation as an XML Document and construct Agent Information from an XML Document 0.2.0 - Moved getFieldElement to WorkslipSerializer.serializeWorkslipField and inflateWorkslipField to WorkslipSerializer.inflateWorkslipField 0.2.1 - Added contact string (see AgentInformation (in A.1.3, page 301))

See also

- AgentInformation (in A.1.3, page 301)

Declaration

```java
public class AgentInformationSerializer
    extends java.lang.Object
```

Constructor summary

```java
AgentInformationSerializer()
```

Method summary

- `xmlInflate(File)` Produces an AgentInformation document with values extracted from the XML document described in the file argument.

- `xmlSerialize(AgentInformation)` Serializes agent information into an XML document.
Constructors

- AgentInformationSerializer

    public AgentInformationSerializer()

Methods

- xmlInflate

    public static AgentInformation xmlInflate(java.io.File file) throws JDOMException, java.io.IOException, java.lang.ClassNotFoundException

    - Description

        Produces an AgentInformation document with values extracted from the XML document described in the file argument.

    - Parameters

        * file – the xml document to be loaded into an AgentInformation (in A.1.3, page 301)

    - Returns – the agent information described in the file

    - Throws

        * JDOMException – if there is a problem extracting information from the document

        * java.io.IOException – if there is a problem extracting information from the file
A.1. PACKAGE PERSONALAGENTS

* java.lang.ClassNotFoundException – if there is a problem loading the JDOM xml tools.

• xmlSerialize

    public static java.lang.String xmlSerialize(AgentInformation agentInformation)

    – Description

    Serializes agent information into an XML document.

    – Parameters

    * agentInformation – the agent information to serialize

    – Returns – string containing the xml document describing the agent information

A.1.5 Class AgentResources

Provides common resources that may be needed thought an agent. It provides access to the configuration and master workslip as well as convenience methods and variables to manage agent work.

Declaration

public class AgentResources
    extends java.lang.Object

Field summary

**AGENT_TOUCH_FIELD** Special field used by all agents to note when they have worked on a workslip.

**INPUT_WORKSLIP_FILE_LOCATION_CONFIG** Configuration to determine where the agent will read workslips from.

**OUTPUT_WORKSLIP_FILE_LOCATION_CONFIG** Configuration to determine where the agent will write workslips to.

Constructor summary

AgentResources()

Method summary

**addConfigurations(ConfigurationSet)** Adds a ConfigurationSet (in A.6.1, page 351) to the common configuration, adding new values.

**getAgentInformation()** Allow components within the Cell access information about the agent.

**getAgentName()** Convenience method to retrieve agent name from the AgentInformation (in A.1.3, page 301).

**getConfiguration(String)** Returns the configuration if it exists or null otherwise.

**getConfigurationIfExists(String)** Returns the configuration if it exists or an empty string otherwise.

**getConfigurationSet()** Returns the common configuration that the agent will use to access the values it needs to operate.
getFieldsListed(String) Parses a list of workslip fields from a string.

getMasterWorkslip() Returns the master workslip for the agent.

getObjectDataSerializer() Returns the common ObjectDataSerializer (in A.10.1, page 406) for the agent to use to serialize objects into a format to be passed to other agents.

sanitizeStringForWorkslipFieldName(String) Tidies a string to make it usable as a workslip name.

setAgentInformation(AgentInformation) Allow the Cell to set the agent information for other components to access.

setConfiguration(ConfigurationSet) Sets the common configuration that the agent will use to access the values it needs to operate.

setMasterWorkslip(MasterWorkslip) Sets the master workslip for the agent to use.

setObjectDataSerializer(ObjectDataSerializer) Sets the common ObjectDataSerializer (in A.10.1, page 406) for the agent to use to serialize objects into a format to be passed to other agents.

touchWorkslip(Workslip, AgentInformation) Convenience method to update the special touch field in the workslip to indicate this agent has read the workslip.

Fields

• public static final java.lang.String INPUT_WORKSLIP_FILE_LOCATION_CONFIG – Configuration to determine where the agent will read workslips from

• public static final java.lang.String OUTPUT_WORKSLIP_FILE_LOCATION_CONFIG
- Configuration to determine where the agent will write workslips to

  - public static final workslip.WorkslipField AGENT_TOUCH_FIELD

    - Special field used by all agents to note when they have worked on a workslip.

Constructors

- AgentResources

  public AgentResources()

Methods

- addConfigurations

  public static void addConfigurations(configuration.ConfigurationSet newConfiguration)

  - Description

    Adds a ConfigurationSet (in A.6.1, page 351) to the common configuration, adding
    new values. This will overwrite any previous values in the configuration.

  - Parameters

    * newConfiguration – the configuration to add to the configuration set.

- getAgentInformation
public static AgentInformation getAgentInformation()

- **Description**
  Allow components within the Cell access information about the agent
- **Returns** – returns the agent information for this agent.

- **getAgentName**

public static java.lang.String getAgentName()

- **Description**
  Convenience method to retrieve agent name from the AgentInformation (in A.1.3, page 301).
- **Returns** – the agent name as set in the common agent information

- **getConfiguration**

public static java.lang.String getConfiguration(java.lang.String configurationKey)

- **Description**
  Returns the configuration if it exists or null otherwise.
- **Parameters**
  * configurationKey – the key of the configuration to check for
• getConfigurationIfExists

```java
public static java.lang.String getConfigurationIfExists(java.lang.String configurationKey)
```

- **Description**
  Returns the configuration if it exists or an empty string otherwise.

- **Parameters**
  - * configurationKey – the key of the configuration to check for

- **Returns** – the value of the configuration if possible or the empty string otherwise.

• getConfigurationSet

```java
public static configuration.ConfigurationSet getConfigurationSet()
```

- **Description**
  Returns the common configuration that the agent will use to access the values it needs to operate.

- **Returns** – returns the `ConfigurationSet` (in A.6.1, page 351) for the Agent.

• getFieldsListed

```java
public static java.util.List getFieldsListed(java.lang.String fieldsString)
```
- **Description**
  Pares a list of workslip fields from a string. Each field separated with a ‘!’ character. Parsing the fields from the string is done with the *WorkslipField* (in A.10.5, page 422) string constructor.

- **Parameters**
  
  * fieldsString – a list of workslip fields, separated by ‘!’

- **Returns** – a list of the workslip fields in the string

- **getMasterWorkslip**

  ```java
  public static masterworkslip.MasterWorkslip getMasterWorkslip()
  ```

  - **Description**
    Returns the master workslip for the agent. This returns a list of all the fields the agent knows about.

  - **Returns** – the master workslip for the agent.

- **getObjectDataSerializer**

  ```java
  public static workslip.ObjectDataSerializer getObjectDataSerializer()
  ```

  - **Description**
    Returns the common *ObjectDataSerializer* (in A.10.1, page 406) for the agent to use to serialize objects into a format to be passed to other agents.
- **Returns** – the common ObjectDataSerializer or the default ODS if no other has been set.

- **sanitizeStringForWorkslipFieldName**

  ```java
  public static java.lang.String sanitizeStringForWorkslipFieldName(java.lang.String source)
  ```

  - **Description**
    Tidies a string to make it usable as a workslip name. In particular, trims all leading and trailing whitespace, converts all internal white space to underscores and transforms all characters to lower case.

  - **Parameters**
    * **source** – the base string

  - **Returns** – the string sanitized as above.

- **setAgentInformation**

  ```java
  public static void setAgentInformation(AgentInformation agentInformation)
  ```

  - **Description**
    Allow the Cell to set the agent information for other components to access.

  - **Parameters**
    * **agentInformation** – the AgentInformation (in A.1.3, page 301) to store for the agent
A.1. PACKAGE PERSONALAGENTS

- setConfiguration

```java
public static void setConfiguration(configration.ConfigurationSet config)
```

- Description

Sets the common configuration that the agent will use to access the values it needs to operate.

- Parameters

  * `config` – the configuration to store for the agent.

- setMasterWorkslip

```java
public static void setMasterWorkslip(masterworkslip.MasterWorkslip masterWorkslip)
```

- Description

Sets the master workslip for the agent to use.

- Parameters

  * `masterWorkslip` – the master workslip to be used by the agent.

- setObjectDataSerializer

```java
public static void setObjectDataSerializer(workslip.ObjectDataSerializer ods)
```
A.2. PACKAGE PERSONALAGENTS.AGENTUTILITIES

- **Description**

  Sets the common ObjectDataSerializer (in A.10.1, page 406) for the agent to use to serialize objects into a format to be passed to other agents.

- **Parameters**

  * `ods` – the Serializer to set as the common Serializer.

- **touchWorkslip**

  public static workslip.Workslip touchWorkslip(workslip.Workslip workslip, AgentInformation agentInformation)

  - **Description**

    Convenience method to update the special touch field in the workslip to indicate this agent has read the workslip.

  - **Parameters**

    * `workslip` – the workslip to ”stamp”

    * `agentInformation` – the agent information for the agent

  - **Returns** – the workslip having the touch field updated

### A.2 Package personalagents.agentutilities

*Package Contents Page*

Classes

**FloatExtractor** ................................................................. 319
Utility class to help extract float values from a `WorkslipField` (in A.10.5, page 422).

### A.2.1 Class FloatExtractor

Utility class to help extract float values from a `WorkslipField` (in A.10.5, page 422).

**Declaration**

```java
class FloatExtractor
extends java.lang.Object
```

**Constructor summary**

```java
FloatExtractor()
```

**Method summary**

```java
extractFloat(Workslip, WorkslipField) Extracts a float value from the field value of the given field.
```

**Constructors**

- `FloatExtractor`

```java
public FloatExtractor()
```

**Methods**

- `extractFloat`
public static float extractFloat(personalagents.workslip.Workslip w,personalagents.workslip.WorkslipField f)

- **Description**
  Extracts a float value from the field value of the given field. If it exists within the given workslip and can reasonably be converted to a float, it is returned. Any numeric type should work, as should a string encoding a float. If a float cannot be parsed an IllegalArgumentException is thrown.

- **Parameters**
  
  * `w` – the workslip to take the data from
  * `f` – the field to parse a float value from

- **Returns** – if possible the value of the field as a float.

### A.3 Package personalagents.cell

*Package Contents*  

**Interfaces**

- **AgentClassFactory** .................................................................321
  Dynamic loader for agents.

**Classes**

- **AgentCell** .................................................................322
  The AgentCell provides the infrastructure to execute the agent, set up the configuration, access resources and manage communication.
A.3. PACKAGE PERSONALAGENTS.CELL

ReflectiveAgentClassFactory.................................326

Dynamic agent loader using Java’s core reflection system.

A.3.1 Interface AgentClassFactory

Dynamic loader for agents.

Declaration

public interface AgentClassFactory

All known subinterfaces

ReflectiveAgentClassFactory (in A.3.3, page 326)

All classes known to implement interface

ReflectiveAgentClassFactory (in A.3.3, page 326)

Method summary

getAgentClassByName(String) Dynamically loads an agent using its class name.

Methods

• getAgentClassByName

personalagents.Agent getAgentClassByName(java.lang.String agentClassName)
A.3. PACKAGE PERSONALAGENTS.CELL

- Description
  Dynamically loads an agent using its class name.

- Parameters
  * `agentClassName` – the fully qualified name of the agent class.

- Returns
  an Agent instance of the class named.

A.3.2 Class AgentCell

The AgentCell provides the infrastructure to execute the agent, set up the configuration, access resources and manage communication.

Declaration

```java
public class AgentCell
    extends java.lang.Object
```

Constructor summary

- `AgentCell(Agent, AgentExecutionHandler)` Create the new cell to supply, organize and execute the agent.

Method summary

- `getAgent()` Returns the agent proper.
- `getExecutionModel()` Returns the agent execution model.
- `loadConfig(ConfigurationSet)` Loads the configuration for the Agent.
- `main(String[])` Sets up and creates the cell which then runs the agent.
- `runAgent()` Execute the agent with the given execution model
shutdownAgent() Shutdown stub.

Constructors

- AgentCell

```java
public AgentCell(personalagents.Agent agent,
                 executionmodel.AgentExecutionHandler executionModel)
```

- Description
  Create the new cell to supply, organize and execute the agent.

- Parameters
  * `agent` - the agent which this cell will execute
  * `executionModel` - the model which will execute the agent

Methods

- getAgent

```java
public personalagents.Agent getAgent()
```

- Description
  Returns the agent proper.

- Returns - the `Agent` (in A.1.1, page 297).

- getExecutionModel
A.3. PACKAGE PERSONALAGENTS.CELL

public executionmodel.AgentExecutionHandler getExecutionModel()

- **Description**
  Returns the agent execution model.

- **Returns** – the `AgentExecutionHandler` (in A.4.1, page 328) that will run this agent.

- **loadConfig**

    public static personalagents.configuration.ConfigurationSet
    loadConfig(personalagents.configuration.ConfigurationSet
    argumentConfigs) throws java.io.IOException

    - **Description**
      Loads the configuration for the Agent. Takes the argument configurations which
      override the configuration file configurations, which override the the agent’s de-
      fault configurations. The file configuration will look for the configuration file
      property in the argument configurations and override the default configuration
      file location.

    - **Parameters**
      *
      - `argumentConfigs` – the configurations collected from the commandline

    - **Returns** – the full set of configurations from the agent

    - **Throws**
      *
      - `java.io.IOException` – if it is not possible to load configurations from the
        file
• main

```
public static void main(java.lang.String[] args)
```

– Description
Sets up and creates the cell which then runs the agent. Also sets the common cell configuration.

– Parameters
  * `args` – list of arguments, will be parsed as configuration values

• runAgent

```
public void runAgent()
```

– Description
Execute the agent with the given execution model

• shutdownAgent

```
public void shutdownAgent()
```

– Description
Shutdown stub. Has no use in this version.
A.3.3 Class ReflectiveAgentClassFactory

Dynamic agent loader using Java’s core reflection system.

Declaration

```java
public class ReflectiveAgentClassFactory
    extends java.lang.Object implements AgentClassFactory
```

Constructor summary

```java
ReflectiveAgentClassFactory()
```

Method summary

```java
getAgentClassByName(String) Dynamically loads an agent with the given
    fully qualified name.
```

Constructors

- ReflectiveAgentClassFactory

```java
public ReflectiveAgentClassFactory()
```

Methods

- getAgentClassByName
public personalagents.Agent getAgentClassByName(java.lang.String agentClassName)

- Description
Dynamically loads an agent with the given fully qualified name. Agents are loaded via the Java Reflection API. In order to succeed the JVM must have access to the class for the agent and the agent class must have a available no argument constructor. In the event that the reflection api cannot load the class for the agent the agent instance will be returned as null and the error will be logged.

- Parameters
  * agentClassName – the fully qualified name for the agent class

- Returns – an agent of given class, if successful, null otherwise

A.4 Package personalagents.cell.executionmodel

Package Contents

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentExecutionHandler</td>
<td>328</td>
</tr>
<tr>
<td>An approach to executing an agent.</td>
<td></td>
</tr>
<tr>
<td>AgentExecutionHandlerFactory</td>
<td>330</td>
</tr>
<tr>
<td>Factory interface to define how to get an AgentExecutionModel.</td>
<td></td>
</tr>
<tr>
<td>RegistrationHandler</td>
<td>331</td>
</tr>
<tr>
<td>Registers the agent with the given AgentInformation (in A.1.3, page 301) with the correct outside partner, be that other agents or a dispatcher or some other service.</td>
<td></td>
</tr>
</tbody>
</table>
A.4. PACKAGE PERSONALAGENTS.CELL.EXECUTIONMODEL

Classes

ContinuousFilePassingExecutionHandler .............................................. 332
Execution Handler which runs an agent repeated until it is stopped.
EmptyWorkslipOutputExecutionHandler ............................................ 334
An Execution Handler which will run an agent once and requesting it
to output an empty workslip to its output directory.
ExecutionHandlerFactory ................................................................. 337

FileWritingExecutionHandler ............................................................ 339
Base Execution Handler for the file writing family.
FileWritingRegistrationHandler ......................................................... 342
Registration Handler for agents which write registration information to
file.
OneRunFilePassingExecutionHandler ................................................. 343
FileWritingExecutionModel implementation which runs once and stops.
RegistrationExecutionHandler .......................................................... 346
Runs the agent in registration mode to register itself then shuts down.
VersionExecutionHandler ............................................................... 347
Runs an agent in version mode, and write useful information about the
agent to the console

A.4.1 Interface AgentExecutionHandler

An approach to executing an agent.

Declaration

public interface AgentExecutionHandler
All known subinterfaces


All classes known to implement interface


Method summary

run(Agent) Executes the agent in the manner needed for this handler.

Methods

• run

void run(personalagents.Agent agent)

- Description
  
  Executes the agent in the manner needed for this handler.

- Parameters

  * agent – the agent to be executed.
A.4.2 Interface AgentExecutionHandlerFactory

Factory interface to define how to get an AgentExecutionModel.

Declaration

public interface AgentExecutionHandlerFactory

All known subinterfaces

ExecutionHandlerFactory (in A.4.6, page 337)

All classes known to implement interface

ExecutionHandlerFactory (in A.4.6, page 337)

Method summary

getExecutionModel(String) Dynamically returns an agentExecutionModel with the given name.

Methods

- getExecutionModel

    AgentExecutionHandler getExecutionModel(java.lang.String executionModelName)

    - Description
      Dynamically returns an agentExecutionModel with the given name.

    - Parameters
A.4.3 Interface RegistrationHandler

Registers the agent with the given AgentInformation (in A.1.3, page 301) with the correct outside partner, be that other agents or a dispatcher or some other service.

Declaration

public interface RegistrationHandler

All known subinterfaces

FileWritingRegistrationHandler (in A.4.8, page 342)

All classes known to implement interface

FileWritingRegistrationHandler (in A.4.8, page 342)

Method summary

register(AgentInformation)

Methods

• register

    boolean register(personalagents.AgentInformation agentinformation)
A.4.4 Class ContinuousFilePassingExecutionHandler

Execution Handler which runs an agent repeated until it is stopped.

Declaration

```java
public class ContinuousFilePassingExecutionHandler
    extends personalagents.cell.executionmodel.FileWritingExecutionHandler
```

Constructor summary

- `ContinuousFilePassingExecutionHandler()` Creates a new `ContinuousFilePassingExecutionHandler` and configures it from the `AgentResources` (in A.1.5, page 309) configuration set.

Method summary

- `getLogIdentifier()` Returns an identifier for this class which can be used in the log system.
- `getOutputDirectory()` Returns the directory where this agent will write its output.
- `run(Agent)` Executes the given agent at the configured interval.

Constructors

- `ContinuousFilePassingExecutionHandler()`
A.4. PACKAGE PERSONALAGENTS.CELL.EXECUTIONMODEL

- Description

Creates a new ContinuousFilePassingExecutionHandler and configures it from the AgentResources (in A.1.5, page 309) configuration set. This includes the intake and output directories where it will read and write workslips, as well as the agent loop time.

Methods

- getLogIdentifier

protected java.lang.String getLogIdentifier()

  - Description

    Returns an identifier for this class which can be used in the log system.

  - Returns – an identifier for this class.

- getOutputDirectory

protected java.lang.String getOutputDirectory()

  - Description

    Returns the directory where this agent will write its output.

  - Returns – the directory name where this agent will output.

- run
public void run(personalagents.Agent agent)

- Description
  Executes the given agent at the configured interval. Agent is executed in a new thread through the Timer.

- Parameters
  * agent – the agent to be executed

Members inherited from class FileWritingExecutionHandler

personalagents.cell.executionmodel.FileWritingExecutionHandler (in A.4.7, page 339)

- protected abstract String getLogIdentifier()
- protected abstract String getOutputDirectory()
- protected Collection getWorkslipFiles(java.io.File intakeDirectory)
- protected void processWorkslipFromFile(personalagents.Agent agent, java.io.File intakeDirectory)

A.4.5 Class EmptyWorkslipOutputExecutionHandler

An Execution Handler which will run an agent once and requesting it to output an empty workslip to its output directory.

Declaration

public class EmptyWorkslipOutputExecutionHandler
    extends personalagents.cell.executionmodel.FileWritingExecutionHandler
Constructor summary

EmptyWorkslipOutputExecutionHandler() Creates a new EmptyWorkslipOutputExecutionHandler, with its output directory set from the AgentResources (in A.1.5, page 309) configuration Set.

Method summary

getLogIdentifier() Returns the unique identifier for this class for logging purposes.

getOutputDirectory() Returns the output directory for this execution handler.

outputEmptyWorkslipFile() Creates an empty workslip and writes it to the target directory.

run(Agent) Runs the given agent generates an empty workslip file

Constructors

- EmptyWorkslipOutputExecutionHandler

public EmptyWorkslipOutputExecutionHandler()

  Description

  Creates a new EmptyWorkslipOutputExecutionHandler, with its output directory set from the AgentResources (in A.1.5, page 309) configuration Set.

Methods

- getLogIdentifier
protected java.lang.String getLogIdentifier()

- **Description**
  
  Returns the unique identifier for this class for logging purposes.
  
- **Returns** – returns an identifier to be used for the log.

• **getOutputDirectory**

protected java.lang.String getOutputDirectory()

- **Description**
  
  Returns the output directory for this execution handler.
  
- **Returns** – the location where this agent will write its files

• **outputEmptyWorkslipFile**

protected void outputEmptyWorkslipFile()

- **Description**
  
  Creates an empty workslip and writes it to the target directory.

• **run**

    public void run(personalagents.Agent agent)
A.4. PACKAGE PERSONALAGENTS.CELL.EXECUTIONMODEL

- Description
  
  Runs the given agent generates an empty workslip file

- Parameters
  
  * agent – the agent to be executed

Members inherited from class FileWritingExecutionHandler

personalagents.cell.executionmodel.FileWritingExecutionHandler (in A.4.7, page 339)

- protected abstract String getLogIdentifier()
- protected abstract String getOutputDirectory()
- protected Collection getWorkslipFiles(java.io.File intakeDirectory)
- protected void processWorkslipFromFile(personalagents.Agent agent, java.io.File intakeDirectory)

A.4.6 Class ExecutionHandlerFactory

Declaration

public class ExecutionHandlerFactory
  
  extends java.lang.Object implements AgentExecutionHandlerFactory

Constructor summary

  ExecutionHandlerFactory()
Method summary

**getExecutionModel(String)** Returns the AgentExecutionHandler (in A.4.1, page 328) that corresponds to the given name.

Constructors

- **ExecutionHandlerFactory**

  ```java
  public ExecutionHandlerFactory()
  ```

Methods

- **getExecutionModel**

  ```java
  public AgentExecutionHandler getExecutionModel(java.lang.String executionModelName)
  ```

  - **Description**
    
    Returns the AgentExecutionHandler (in A.4.1, page 328) that corresponds to the given name.

  - **Parameters**
    
    - `executionModelName` – the name of the given execution model.

  - **Returns** – the agent execution model.
A.4.7 Class FileWritingExecutionHandler

Base Execution Handler for the file writing family. Runs agents and communicates with other agents by writing workslips as files in common locations. Does the same with registration information.

Declaration

public abstract class FileWritingExecutionHandler
    extends java.lang.Object implements AgentExecutionHandler

All known subclasses


Constructor summary

    FileWritingExecutionHandler()

Method summary

    getLogIdentifier() Returns an identifier for the execution handler for purposes of logging.

    getOutputDirectory() Returns the location where the agent will write its output.

    getWorkslipFiles(File) Searches at the given location for all files ending in .workslip.
processWorkslipFromFile(Agent, File) Execute an agent to process a workslip from a file.

Constructors

• FileWritingExecutionHandler

public FileWritingExecutionHandler()

Methods

• getLogIdentifier

protected abstract java.lang.String getLogIdentifier()

– Description
Returns an identifier for the execution handler for purposes of logging.

– Returns – an identifier for this execution handler to use for the log.

• getOutputDirectory

protected abstract java.lang.String getOutputDirectory()

– Description
Returns the location where the agent will write its output.

– Returns – the file system path to the location where the agent can write.
• getWorkslipFiles

protected java.util.Collection getWorkslipFiles(java.io.File intakeDirectory)

− Description

Searches at the given location for all files ending in .workslip. All matching files are returned in the Collection of Files.

− Parameters

  * intakeDirectory -- the directory to check for files

− Returns – a collection of Files for all .workslip files

• processWorkslipFromFile

protected void processWorkslipFromFile(personalagents.Agent agent, java.io.File intakeDirectory)

− Description

Execute an agent to process a workslip from a file. Will draw the workslips from the intake directory and feed them to the agent.

− Parameters

  * agent -- the agent handling the processing

  * intakeDirectory -- the direction to read workslip files from.
A.4.8 Class FileWritingRegistrationHandler

Registration Handler for agents which write registration information to file. Allows the agent to communicate their read and updated fields, with other agents (primarily the dispatcher)

Declaration

```
public class FileWritingRegistrationHandler
    extends java.lang.Object implements RegistrationHandler
```

Constructor summary

```
FileWritingRegistrationHandler()
```

Method summary

```
register(AgentInformation) Registers the agent with the provided AgentInformation
    (in A.1.3, page 301).
```

Constructors

- FileWritingRegistrationHandler

```
public FileWritingRegistrationHandler()
```

Methods

- register
public boolean register(personalagents.AgentInformation agentInformation)

- **Description**
  Registers the agent with the provided `AgentInformation` (in A.1.3, page 301). Produces an agent registration file in the configured registration location. The file contains the XML serialization of the `AgentInformation` as produced by the `AgentInformationSerializer` (in A.1.4, page 307).

- **Parameters**
  * `agentInformation` – the information of the agent to be registered

- **Returns** – true if the file was successfully generated and saved. False if an error occurred. (Message logged to the agent log).

### A.4.9 Class OneRunFilePassingExecutionHandler

FileWritingExecutionModel implementation which runs once and stops.

**Declaration**

```java
public class OneRunFilePassingExecutionHandler
    extends personalagents.cell.executionmodel.FileWritingExecutionHandler
```

**Constructor summary**

**OneRunFilePassingExecutionHandler()** Creates a new OneRunFilePassingExecutionHandler, which is configured with the input and output directories from the agent configuration.
Method summary

getLogIdentifier() Returns an identifier for this class for logging purposes.

getOutputDirectory() Returns the agent’s output directory.

run(Agent) Processes one workslip from the agent’s intake directory and then halts.

Constructors

• OneRunFilePassingExecutionHandler

public OneRunFilePassingExecutionHandler()

— Description

Creates a new OneRunFilePassingExecutionHandler, which is configured with the input and output directories from the agent configuration.

Methods

• getLogIdentifier

protected java.lang.String getLogIdentifier()

— Description

Returns an identifier for this class for logging purposes.

— Returns – an identifier for this class for the log.
A.4. PACKAGE PERSONALAGENTS.CELL.EXECUTIONMODEL

- **getOutputDirectory**

  protected java.lang.String getOutputDirectory()

  - **Description**

    Returns the agent’s output directory.

  - **Returns** – the directory the agent will write files to.

- **run**

  public void run(personalagents.Agent agent)

  - **Description**

    Processes one workslip from the agent’s intake directory and then halts.

  - **Parameters**

    * agent – the agent to execute

Members inherited from class FileWritingExecutionHandler

personalagents.cell.executionmodel.FileWritingExecutionHandler  (in A.4.7, page 339)

- protected abstract String getLogIdentifier()
- protected abstract String getOutputDirectory()
- protected Collection getWorkslipFiles(java.io.File intakeDirectory)
- protected void processWorkslipFromFile(personalagents.Agent agent, java.io.File intakeDirectory)
A.4. Class RegistrationExecutionHandler

Runs the agent in registration mode to register itself then shuts down.

Declaration

```java
public class RegistrationExecutionHandler
extends java.lang.Object implements AgentExecutionHandler
```

Constructor summary

```java
RegistrationExecutionHandler(RegistrationHandler)
```

- Creates a new RegistrationExecutionHandler with the given registration model, which it will use to register the agent.

Method summary

```java
run(Agent)
```

- Runs the instance registration model with the information for the provided agent.

Constructors

- RegistrationExecutionHandler

```java
public RegistrationExecutionHandler(RegistrationHandler registrationModel)
```

- Description

  Creates a new RegistrationExecutionHandler with the given registration model, which it will use to register the agent.
- Parameters
  
  * registrationModel – the model to use to register the agent.

Methods

- run

  public void run(personalagents.Agent agent)

  - Description
    
    Runs the instance registration model with the information for the provided agent.

  - Parameters
    
    * agent – the agent to be registered.

A.4.11 Class VersionExecutionHandler

Runs an agent in version mode, and write useful information about the agent to the console

Declaration

public class VersionExecutionHandler
  extends java.lang.Object implements AgentExecutionHandler

Constructor summary

  VersionExecutionHandler()
Method summary

run(Agent) Produces information about the provided agent on the console.

Constructors

• VersionExecutionHandler

    public VersionExecutionHandler()

Methods

• run

    public void run(personalagents.Agent agent)

    – Description
      Produces information about the provided agent on the console. The version mode
      configuration is consulted to determine how much information to provide.

    – Parameters
      * agent – the agent to provide information about
A.5. **PACKAGE PERSONALAGENTS.COMPOSABLEAGENT**

**InputHandler** ................................................................. 349

**Processor** ........................................................................... 349

**Classes**

**ComposableAgent** .............................................................. 350

### A.5.1 Interface InputHandler

**Declaration**

```java
public interface InputHandler
```

**Method summary**

- `acceptInput(Workslip)`

**Methods**

- `acceptInput`

  ```java
  void acceptInput(personalagents.workslip.Workslip workslip)
  ```

### A.5.2 Interface Processor

**Declaration**


public interface Processor

Method summary

    process()

Methods

    • process

    void process()

A.5.3 Class ComposableAgent

Declaration

public abstract class ComposableAgent
    extends java.lang.Object implements personalagents.Agent

Constructor summary

    ComposableAgent()

Constructors

    • ComposableAgent

    public ComposableAgent()
A.6  Package personalagents.configuration

ConfigurationSet holds the configuration for an Agent (and Cell).

A.6.1  Class ConfigurationSet

ConfigurationSet holds the configuration for an Agent (and Cell). Each configuration is held as a key value pair. Configurations can be loaded from an array of string, as arguments, or from a file. In the array format each key value pair must be of the form -key=value. In file format the the form is "key pair". Configurations can be set at creation or set individually or merged in from another configuration set. In the instance where configurations are merged (setConfigurations) the configurations from the argument *overwrite* the values in this configuration. ConfigurationSet uses a Java Properties object to hold the values. This allows the system to take advantage of the existing reading and writing systems for Properties objects.

Declaration

```java
public class ConfigurationSet
    extends java.lang.Object
```
Field summary

**ARGUMENT_REGEX_PATTERN** The pattern for configurations that are held as one string in the –key=value pair format.

Constructor summary

**ConfigurationSet()** Creates an empty configuration set.

**ConfigurationSet(File)** Creates a new ConfigurationSet and fills it with the configurations held in the given file.

**ConfigurationSet(String[])** Creates new ConfigurationSet and fills it with configurations taken from the array.

Method summary

**getConfiguration(String)** Returns the value associated with the given key, or throws a MissingConfigurationException if the configuration is not set in the ConfigurationSet.

**hasConfiguration(String)** Tests if the key is found in this ConfigurationSet.

**setConfiguration(String, String)** Sets the configuration key value pair.

**setConfigurations(ConfigurationSet)** Sets all configurations found in the given ConfigurationSet in this Configuration.

**storeToFile(File, String)** Saves the ConfigurationSet to a file.

Fields

- public static final java.lang.String **ARGUMENT_REGEX_PATTERN**
The pattern for configurations that are held as one string in the key=value pair format.

Constructors

• ConfigurationSet

public ConfigurationSet()

- Description

Creates an empty configuration set.

• ConfigurationSet

public ConfigurationSet(java.io.File configurationFile) throws java.io.FileNotFoundException, java.io.IOException

- Description

Creates a new ConfigurationSet and fills it with the configurations held in the given file. Each configuration must be in the form key value on each line of the file. Defers the loading to the Properties.load(FileReader) method.

- Parameters

  * configurationFile – the file handle for the configuration file

- Throws

  * java.io.FileNotFoundException – if the file cannot be found
A.6. PACKAGE PERSONALAGENTS.CONFIGURATION

* java.io.IOException – if another IO problem occurs

- ConfigurationSet

    public ConfigurationSet(java.lang.String[] args)

    - Description
      Creates new ConfigurationSet and fills it with configurations taken from the array.
      Configurations are taken so long as they are in the format –key=value.

    - Parameters
      * args – the configuration strings in the form –key=value

Methods

- getConfiguration

    public java.lang.String getConfiguration(java.lang.String key) throws
    personalagents.configuration.MissingConfigurationException

    - Description
      Returns the value associated with the given key, or throws a MissingConfigurationException if the configuration is not set in the ConfigurationSet

    - Parameters
      * key – the name of the property

    - Returns – the matched value if it exists
A.6. PACKAGE PERSONALAGENTS.CONFIGURATION

- **Throws**
  
  * personalagents.configuration.MissingConfigurationException – if the configuration is not set

- **hasConfiguration**

  ```
  public boolean hasConfiguration(java.lang.String key)
  ```

  - **Description**
    Tests if the key is found in this ConfigurationSet.

  - **Parameters**
    * key – the configuration to test

  - **Returns** – true if the configuration has a value, false otherwise

- **setConfiguration**

  ```
  public void setConfiguration(java.lang.String key, java.lang.String value)
  ```

  - **Description**
    Sets the configuration key value pair.

  - **Parameters**
    * key – the name of the configuration
    * value – the value of the configuration
• setConfigurations

public void setConfigurations(ConfigurationSet configuration)

  – Description
  Sets all configurations found in the given ConfigurationSet in this Configuration.
  This means that any conflicts are resolved in favour of the given ConfigurationSet.

  – Parameters
  * configuration – the ConfigurationSet to copy into this one.

• storeToFile

public void storeToFile(java.io.File configurationFile, java.lang.String comment) throws java.io.IOException

  – Description
  Saves the ConfigurationSet to a file. Defers the saving to Properties.store(FileWriter).

  – Parameters
  * configurationFile – The file handle to store to
  * comment – a description of the ConfigurationSet

  – Throws
  * java.io.IOException – if an IO problem occurs
A.6.2 Exception MissingConfigurationException

Exception to be thrown when a value cannot be found for a given key in the configuration. Update - Version 0.2.0: changed to extend RuntimeException so that the exception is unchecked.

Declaration

public class MissingConfigurationException
    extends java.lang.RuntimeException

Constructor summary

    MissingConfigurationException(String)

Constructors

- MissingConfigurationException

    public MissingConfigurationException(java.lang.String key)

Members inherited from class Throwable

java.lang.Throwable

- public final synchronized void addSuppressed(Throwables arg0)
- public synchronized Throwable fillInStackTrace()
- public synchronized Throwable getCause()
- public String getLocalizedMessage()
A.7 Package personalagents.dispatcher

Package Contents

Interfaces

RegistrationHandler

A RegistrationHandler takes registration information from agents starting up and returns that information to the rest of the dispatcher.

RegistrationHandlerFactory

Returns a new registration handler with the given name connected to the rest of dispatcher using the given registration queue.

ShowAlive

Periodically prints a mark to the console to demonstrate that the dispatcher is functioning.

WorkslipArchive

Archives the workslips the dispatcher has seen.

WorkslipIntakeHandler

Handler to takes workslips and return it to the dispatcher.
**A.7. PACKAGE PERSONALAGENTS.DISPATCHER** 359

**WorkslipIntakeHandlerFactory** ................................................................. 367

Returns a new `WorkslipIntakeHandler` (in A.7.5, page 366) derived from

the given name.

**WorkslipOutputHandler** ................................................................. 368

Handler to take workslips from the dispatcher and send them to the
given agent.

**WorkslipOutputHandlerFactory** ................................................................. 369

* Returns a new `WorkslipOutputHandler` (in A.7.7, page 368) derived

from the given name.

**Classes**

**DispatcherCell** ................................................................. 371

Cell to load and execute a dispatcher.

**DispatcherTestRunner** ................................................................. 372

**DispatchTesting** ................................................................. 373

**FileReadingRegistrationHandler** ................................................................. 374

* Registration Handler which reads agent registrations as files from a

file system location.

**FileReadingWorkslipIntakeHandler** ................................................................. 376

Takes in Workslips via reading their data out of files.

**FileWritingWorkslipOutputHandler** ................................................................. 378

Workslip Output Handler which shares workslips with other agents as

files.

**FullFileWorkslipArchive** ................................................................. 380

Records a full archive of the workslips the dispatcher sees.

**LightTextWorkslipArchive** ................................................................. 381

Records an archive of the names of the workslips the dispatcher has

seen.
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

**ManualRegistrationHandlerFactory** ............................................. 383
Generates registration handlers based on the handler identification string.

**ManualWorkslipIntakeHandlerFactory** ................................. 384
Generates intake handlers based on the handler identification string.

**ManualWorkslipOutputHandlerFactory** .................................... 386
Generates output handlers based on the handler identification string.

**SilentAlive** ........................................................................... 387
Silent ShowAlive which demonstrates the system is working, but does not actually show anything on the console.

**TickAlive** .............................................................................. 389
Prints a symbol to show that the dispatcher is working.

**WorkslipDispatcher** ................................................................. 391
Core of the Dispatcher, manages all agent registration and workslip routing.

### A.7.1 Interface RegistrationHandler

A **RegistrationHandler** takes registration information from agents starting up and returns that information to the rest of the dispatcher.

**Declaration**

```java
public interface RegistrationHandler
    extends java.langRunnable
```

**All known subinterfaces**

FileReadingRegistrationHandler (in A.7.12, page 374)
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

All classes known to implement interface

FileReadingRegistrationHandler (in A.7.12, page 374)

Method summary

registerAgents() Returns a list of all agents that have sent a registration to
the dispatcher since the last time the method was called.

Methods

• registerAgents

java.util.List registerAgents()

– Description

Returns a list of all agents that have sent a registration to the dispatcher since
the last time the method was called.

– Returns – a list of new agents registering with the dispatcher.

A.7.2 Interface RegistrationHandlerFactory

Returns a new registration handler with the given name connected to the rest of dispatcher
using the given registration queue.

Declaration

public interface RegistrationHandlerFactory
All known subinterfaces


All classes known to implement interface


Method summary

**getDispatcherRegistrationModel(String, BlockingQueue)** Returns a new registration handler identified with the given name and connected to the rest of the dispatcher for communication via the queue.

Methods

- **getDispatcherRegistrationModel**

  RegistrationHandler getDispatcherRegistrationModel(java.lang.String registrationModelName, java.util.concurrent.BlockingQueue agentRegistrationQueue)

  - **Description**
    Returns a new registration handler identified with the given name and connected to the rest of the dispatcher for communication via the queue.

  - **Parameters**
    * registrationModelName – the name of the registration handler
    * agentRegistrationQueue – the queue to communicate with
Returns – the given registration handler or null if the name cannot be resolved.

A.7.3 Interface ShowAlive

Periodically prints a mark to the console to demonstrate that the dispatcher is functioning.

Declaration

public interface ShowAlive

All known subinterfaces

TickAlive (in A.7.21, page 389), SilentAlive (in A.7.20, page 387)

All classes known to implement interface

TickAlive (in A.7.21, page 389), SilentAlive (in A.7.20, page 387)

Method summary

getMark() Returns the mark for this ShowAlive.

printAliveMark() Prints the alive mark to the console.

Methods

• getMark

java.lang.String getMark()
Returns the mark for this ShowAlive.

- **Returns** – the mark to show the dispatcher is alive.

- **printAliveMark**

  ```java
define
```}

\[\text{printAliveMark}\]

- **Description**
  
  Prints the alive mark to the console.

### A.7.4 Interface WorkslipArchive

Archives the workslips the dispatcher has seen.

**Declaration**

```java
public interface WorkslipArchive
```

**All known subinterfaces**

- LightTextWorkslipArchive (in A.7.16, page 381)
- FullFileWorkslipArchive (in A.7.15, page 380)

**All classes known to implement interface**

- LightTextWorkslipArchive (in A.7.16, page 381)
- FullFileWorkslipArchive (in A.7.15, page 380)

**Field summary**

```java
FILE_NAME_FORMAT
```
Method summary

archive(Workslip) Records the given workslip.

getArchives() Creates archives based on the configuration.

Fields

- DateTimeFormatter FILE_NAME_FORMAT

Methods

- archive

  void archive(personalagents.workslip.Workslip w)

  - Description
    Records the given workslip.

  - Parameters
    * w – the workslip to record.

- getArchives

  java.util.List getArchives()

  - Description
    Creates archives based on the configuration. Reads the configuration dispatcher.archive.archivists and creates all archive classes and returns them in the list.
– **Returns** – the list of all archives specified in the dispatcher configuration

### A.7.5 Interface WorkslipIntakeHandler

Handler to takes workslips and return it to the dispatcher.

**Declaration**

```java
public interface WorkslipIntakeHandler
    extends java.langRunnable
```

**All known subinterfaces**

FileReadingWorkslipIntakeHandler (in A.7.13, page 376)

**All classes known to implement interface**

FileReadingWorkslipIntakeHandler (in A.7.13, page 376)

**Method summary**

- `readWorkslips()` Receives workslips from a source and returns all of the workslips received since the last time this method was called.

**Methods**

- `readWorkslips`

  ```java
  java.util.Collection readWorkslips()
  ```
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

- **Description**
  Receives workslips from a source and returns all of the workslips received since the last time this method was called.

- **Returns** – the list of newly seen workslips.

### A.7.6 Interface WorkslipIntakeHandlerFactory

Returns a new `WorkslipIntakeHandler` (in A.7.5, page 366) derived from the given name.

**Declaration**

```java
public interface WorkslipIntakeHandlerFactory
```

**All known subinterfaces**

ManualWorkslipIntakeHandlerFactory (in A.7.18, page 384)

**All classes known to implement interface**

ManualWorkslipIntakeHandlerFactory (in A.7.18, page 384)

**Method summary**

`getWorkslipIntakeModel(String, BlockingQueue)` Returns a `WorkslipIntakeHandler` (in A.7.5, page 366) with the given name, if possible.

**Methods**

- `getWorkslipIntakeModel`
A.7. WORKSLIPINTAKEHANDLER

WorkslipIntakeHandler getWorkslipIntakeModel(java.lang.String intakeModelName, java.util.concurrent.BlockingQueue communicationQueue)

- Description
  Returns a WorkslipIntakeHandler (in A.7.5, page 366) with the given name, if possible. The handler is linked to the rest of the dispatcher with the blocking queue.

- Parameters
  * intakeModelName – the name of the workslip intake handler.
  * communicationQueue – the queue which will be used to pass information to the rest of the dispatcher.

- Returns – the new handler identified by the name, or null if the name cannot be resolved to a handler.

A.7.7 INTERFACE WORKSLIPOUTPUTHANDLER

Handler to take workslips from the dispatcher and send them to the given agent.

Declaration

public interface WorkslipOutputHandler

All known subinterfaces

FileWritingWorkslipOutputHandler (in A.7.14, page 378)
All classes known to implement interface

FileWritingWorkslipOutputHandler (in A.7.14, page 378)

Method summary

outputWorkslip(AgentInformation, Workslip) Takes the given workslip
and sends it to the agent.

Methods

• outputWorkslip

  void outputWorkslip(personalagents.AgentInformation
  ai, personalagents.workslip.Workslip w)

  – Description

  Takes the given workslip and sends it to the agent.

  – Parameters

  * ai – the information for the agent to send the workslip to
  * w – the workslip to send.

A.7.8 Interface WorkslipOutputHandlerFactory

* Returns a new WorkslipOutputHandler (in A.7.7, page 368) derived from the given name.

Declaration
public interface WorkslipOutputHandlerFactory

All known subinterfaces


All classes known to implement interface


Method summary

getWorkslipOutputModel(String) Returns a WorkslipOutputHandler (in A.7.7, page 368) with the given name, if possible.

Methods

• getWorkslipOutputModel

WorkslipOutputHandler getWorkslipOutputModel(String modelName)

– Description

Returns a WorkslipOutputHandler (in A.7.7, page 368) with the given name, if possible.

– Parameters

* modelName – the name of the workslip output handler.

– Returns – the new handler identified by the name, or null if the name cannot be resolved to a handler.
A.7.9 Class DispatcherCell

Cell to load and execute a dispatcher. Provides external resources as needed.

Declaration

```java
public class DispatcherCell
    extends java.lang.Object
```

Field summary

- `fullConfiguration` Full configuration for the dispatcher.

Constructor summary

- `DispatcherCell()`

Method summary

- `main(String[])` Loads, configures and then executes a dispatcher.

Fields

- `public static personalagents.configuration.ConfigurationSet fullConfiguration`
  - Full configuration for the dispatcher.

Constructors

- `DispatcherCell()`
public DispatcherCell()

Methods

• main

public static void main(java.lang.String[] args)

- Description
  Loads, configures and then executes a dispatcher.

- Parameters
  * args – the command line arguments, which will be parsed as part of the dispatcher configuration

A.7.10 Class DispatcherTestRunner

Declaration

public class DispatcherTestRunner
  extends java.lang.Object

Constructor summary

DispatcherTestRunner()

Method summary

main(String[])

Constructors

- DispatcherTestRunner

```java
public DispatcherTestRunner()
```

Methods

- main

```java
public static void main(java.lang.String[] args)
```

### A.7.11 Class DispatchTesting

Declaration

```java
public class DispatchTesting
    extends java.lang.Object
```

Constructor summary

- DispatchTesting()

Method summary

- main(String[])
Constructors

- DispatchTesting

  public DispatchTesting()

Methods

- main

  public static void main(java.lang.String[] args)

A.7.12 Class FileReadingRegistrationHandler

* Registration Handler which reads agent registrations as files from a file system location.

Declaration

public class FileReadingRegistrationHandler
  extends java.lang.Object implements RegistrationHandler

Field summary

- DEFAULT_LOOP_DELAY
- LOOP_DELAY_MS
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

Constructor summary

FileReadingRegistrationHandler(BlockingQueue) Creates a new registration handler which will find agent information by reading it from files.

Method summary

registerAgents() Reads any .agent_registration files in the handler’s configured intake directory and produce AgentInformation (in A.1.3, page 301) for the dispatcher

run() Runs the handler on a timer loop as configured by the dispatcher configuration.

Fields

- public static final int DEFAULT_LOOP_DELAY
- public static final java.lang.String LOOP_DELAY_MS

Constructors

- FileReadingRegistrationHandler

    public FileReadingRegistrationHandler(java.util.concurrent.BlockingQueue agentRegistrationQueue)

    - Description

        Creates a new registration handler which will find agent information by reading it from files.
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

− Parameters

* agentRegistrationQueue – a queue of agent information which is shared between the reading handler and the dispatcher.

Methods

− registerAgents

public java.util.List registerAgents()

− Description

Reads any .agent_registration files in the handler’s configured intake directory and produce AgentInformation (in A.1.3, page 301) for the dispatcher

− Returns – returns a list of agents which have registered with the dispatcher

− run

public void run()

− Description

Runs the handler on a timer loop as configured by the dispatcher configuration.

A.7.13 Class FileReadingWorkslipIntakeHandler

Takes in Workslips via reading their data out of files.
Declaration

public class FileReadingWorkslipIntakeHandler
extends java.lang.Object implements WorkslipIntakeHandler

Field summary

DEFAULT_LOOP_DELAY
LOOP_DELAY_MS
WORKSLIP_INTAKE_DIRECTORY_CONFIG

Configuration entry for the name of the directory where the dispatcher will look for configuration files

Method summary

readWorkslips()

run() Sets up and runs the intake handler to run at regular intervals.

Fields

• public static final java.lang.String WORKSLIP_INTAKE_DIRECTORY_CONFIG
  
  - Configuration entry for the name of the directory where the dispatcher will look for configuration files

• public static final int DEFAULT_LOOP_DELAY

• public static final java.lang.String LOOP_DELAY_MS

Methods

• readWorkslips
java.util.Collection readWorkslips()

- Description copied from WorkslipIntakeHandler (in A.7.5, page 366)
  Receives workslips from a source and returns all of the workslips received since the last time this method was called.

- Returns – the list of newly seen workslips.

• run

public void run()

- Description
  Sets up and runs the intake handler to run at regular intervals.

A.7.14 Class FileWritingWorkslipOutputHandler

Workslip Output Handler which shares workslips with other agents as files.

Declaration

public class FileWritingWorkslipOutputHandler
  extends java.lang.Object implements WorkslipOutputHandler

Constructor summary

FileWritingWorkslipOutputHandler() Creates a FileWritingWorkslipOutputHandler.
Method summary

outputWorkslip(AgentInformation, Workslip) Outputs the workslip as an xml file in the appropriate agent’s intake location.

Constructors

- FileWritingWorkslipOutputHandler

  public FileWritingWorkslipOutputHandler()

    - Description

      Creates a FileWritingWorkslipOutputHandler. Has no configuration because all information is held in the agent information from registration.

Methods

- outputWorkslip

  public void outputWorkslip(personalagents.AgentInformation ai, personalagents.workslip.Workslip w)

    - Description

      Outputs the workslip as an xml file in the appropriate agent’s intake location.

    - Parameters

      * ai – the agent information, which will be used to determine where to place the agent’s workslip file.
A.7.15 Class FullFileWorkslipArchive

Records a full archive of the workslips the dispatcher sees.

Declaration

```java
public class FullFileWorkslipArchive
    extends java.lang.Object implements WorkslipArchive
```

Constructor summary

```java
FullFileWorkslipArchive()
```

Description

Creates a new FullFileWorkslipArchive configured with the location of the archive.

Method summary

```java
archive(Workslip)
```

Records the workslip to the archive location as an xml file.

Constructors

- FullFileWorkslipArchive

```java
public FullFileWorkslipArchive()
```

Description

Creates a new FullFileWorkslipArchive configured with the location of the archive.
Methods

- archive

```java
public void archive(personalagents.workslip.Workslip w)
```

- **Description**
  Records the workslip to the archive location as an xml file. File name is generated based on the time the dispatcher sees the agent and the workslip name.

- **Parameters**

  * w – the th workslip to be recorded

A.7.16 Class LightTextWorkslipArchive

Records an archive of the names of the workslips the dispatcher has seen. The archive is recorded as a file in the archive location.

**Declaration**

```java
public class LightTextWorkslipArchive
    extends java.lang.Object implements WorkslipArchive
```

**Constructor summary**

```java
LightTextWorkslipArchive() Creates a new LightTextWorkslipArchive which is configured with the location of the archive.
```
Method summary

archive(Workslip) Archives a the name of workslip in the archive file.

Constructors

• LightTextWorkslipArchive

    public LightTextWorkslipArchive()

    – Description

    Creates a new LightTextWorkslipArchive which is configured with the location of the archive. Creates a new file ready to be written there.

Methods

• archive

    public void archive(personalagents.workslip.Workslip w)

    – Description

    Archives a the name of workslip in the archive file.

    – Parameters

    * w – the workslip to be archived.
A.7.17 Class ManualRegistrationHandlerFactory

Generates registration handlers based on the handler identification string.

Declaration

```java
public class ManualRegistrationHandlerFactory
    extends java.lang.Object
    implements RegistrationHandlerFactory
```

Constructor summary

```java
ManualRegistrationHandlerFactory()
```

Method summary

```java
getDispatcherRegistrationModel(String, BlockingQueue) Returns a new registration model constructed based on the registration model name.
```

Constructors

- ManualRegistrationHandlerFactory

```java
public ManualRegistrationHandlerFactory()
```

Methods

- getDispatcherRegistrationModel
public RegistrationHandler getDispatcherRegistrationModel(java.lang.String
registrationModelName, java.util.concurrent.BlockingQueue
agentRegistrationQueue)

- **Description**
  Returns a new registration model constructed based on the registration model
  name.

- **Parameters**
  *
  * registrationModelName – the name of the model to generate
  *
  * agentRegistrationQueue – the communication queue the model will use to
    pass registration to the dispatcher

- **Returns** – the appropriate registration handler, or null if the name is not recog-
  nized

### A.7.18 Class ManualWorkslipIntakeHandlerFactory

Generates intake handlers based on the handler identification string.

**Declaration**

```java
public class ManualWorkslipIntakeHandlerFactory
    extends java.lang.Object
    implements WorkslipIntakeHandlerFactory
```

**Constructor summary**

```java
ManualWorkslipIntakeHandlerFactory()
```
Method summary

getWorkslipIntakeModel(String, BlockingQueue) Returns a new intake handler constructed based on the intake handler name.

Constructors

- ManualWorkslipIntakeHandlerFactory

public ManualWorkslipIntakeHandlerFactory()

Methods

- getWorkslipIntakeModel

public WorkslipIntakeHandler getWorkslipIntakeModel(java.lang.String intakeModelName, java.util.concurrent.BlockingQueue communicationQueue)

- Description

Returns a new intake handler constructed based on the intake handler name.

- Parameters

  * intakeModelName – the name of the handler to generate
  * communicationQueue – the queue to communicate with the rest of the dispatcher

- Returns

  the correct intake handler, or null if the name cannot be resolved to a handler
A.7.19 Class ManualWorkslipOutputHandlerFactory

Generates output handlers based on the handler identification string.

Declaration

public class ManualWorkslipOutputHandlerFactory
    extends java.lang.Object implements WorkslipOutputHandlerFactory

Constructor summary

    ManualWorkslipOutputHandlerFactory()

Method summary

    getWorkslipOutputModel(String) Returns a new output handler constructed based on the handler name.

Constructors

    • ManualWorkslipOutputHandlerFactory

    public ManualWorkslipOutputHandlerFactory()

Methods

    • getWorkslipOutputModel
public WorkslipOutputHandler getWorkslipOutputModel(java.lang.String modelName)

- **Description**
  Returns a new output handler constructed based on the handler name.

- **Parameters**
  * modelName – the name of the handler to generate

- **Returns** – the correct output handler, or null if the name cannot be resolved to a handler

### A.7.20 Class SilentAlive

Silent ShowAlive which demonstrates the system is working, but does not actually show anything on the console.

**Declaration**

```java
public class SilentAlive
    extends java.lang.Object implements ShowAlive
```

**Constructor summary**

- **SilentAlive()**

**Method summary**

- **getMark()** Returns the empty string as this mark.
printAliveMark() Does nothing, so that the dispatcher doesn’t make any extra output.

toString() Returns string to show the ShowAlive will produce no output.

Constructors

• SilentAlive

   public SilentAlive()

Methods

• getMark

   public java.lang.String getMark()

   – Description
     Returns the empty string as this mark.

   – Returns –

• printAliveMark

   public void printAliveMark()

   – Description
     Does nothing, so that the dispatcher doesn’t make any extra output.
A.7.1 package personalAgentsdispatcher

- toString

    public java.lang.String toString()

    - Description

        Returns string to show the ShowAlive will produce no output.

    - Returns –

A.7.21 Class TickAlive

Prints a symbol to show that the dispatcher is working.

Declaration

public class TickAlive
    extends java.lang.Object implements ShowAlive

Constructor summary

    TickAlive() Creates a new TickAlive with a default mark of .

    TickAlive(String) Creates a new TickAlive with the given tick as its mark.

Method summary

    getMark() Returns the tick for this ShowAlive.

    printAliveMark() Prints the tick to the console.
Constructors

- **TickAlive**

  ```java
  public TickAlive()
  ```

  **Description**

  Creates a new TickAlive with a default mark of .

- **TickAlive**

  ```java
  public TickAlive(java.lang.String tick)
  ```

  **Description**

  Creates a new TickAlive with the given tick as its mark.

  **Parameters**

  * tick –

Methods

- **getMark**

  ```java
  public java.lang.String getMark()
  ```

  **Description**

  Returns the tick for this ShowAlive.
Returns – the tick for this ShowAlive.

printAliveMark

public void printAliveMark()

– Description
  Prints the tick to the console.

A.7.22 Class WorkslipDispatcher

Core of the Dispatcher, manages all agent registration and workslip routing.

Declaration

public class WorkslipDispatcher
  extends java.lang.Object

Constructor summary

WorkslipDispatcher() Creates a new WorkslipDispatcher with all fields of the dispatcher pointed at new objects which can later be added to.

Method summary

addWorkslipField(WorkslipField) Adds a field to the dispatcher master workslip.

findAgentsInterested(Workslip) Finds all agents which read fields that are updated in the workslip.
findFieldsNeverRead() Finds all of the workslip fields in the Master Workslip that no agent reads.
findFieldsNeverWritten() Finds all of the workslip fields in the Master Workslip that no known agent is able to write.
findFieldsNotInMaster() Find all workslip fields that agents are able to read or write but that the dispatcher does not have in its master workslip.
getAgents() Returns a list of all agents currently registered with the dispatcher.
getMasterWorkslip() Returns the master workslip for the dispatcher, which includes all fields needed for the task.
registerAgent(AgentInformation) Takes the information from the registering agent and adds it to the list of agents and adds the agent read fields and written fields to the field lookup tables.

Constructors

- WorkslipDispatcher

public WorkslipDispatcher()

  - Description

  Creates a new WorkslipDispatcher with all fields of the dispatcher pointed at new objects which can later be added to.

Methods

- addWorkslipField
public void addWorkslipField(personalagents.workslip.WorkslipField field)

- Description

   Adds a field to the dispatcher master workslip.

- Parameters

   * field – the field to add to the master workslip.

• findAgentsInterested

    public java.util.Collection
    
    findAgentsInterested(personalagents.workslip.Workslip workslip)

- Description

    Finds all agents which read fields that are updated in the workslip. Will exclude
    agents which have marked the agent touch field which agents will fill in if they
    do not need to see a workslip a second time.

- Parameters

    * workslip – the workslip to be checked

- Returns – a list of all agents that should be sent the workslip.

• findFieldsNeverRead

    public java.util.List findFieldsNeverRead()
A.7. PACKAGE PERSONALAGENTS.DISPATCHER

- Description
  Finds all of the workslip fields in the Master Workslip that no agent reads.

- Returns – a list of all fields never read.

- findFieldsNeverWritten

public java.util.List findFieldsNeverWritten() {

- Description
  Finds all of the workslip fields in the Master Workslip that no known agent is able to write.

- Returns – a list of all workslip fields that are never written

- findFieldsNotInMaster

public java.util.List findFieldsNotInMaster() {

- Description
  Find all workslip fields that agents are able to read or write but that the dispatcher does not have in its master workslip.

- Returns – a list of all fields that an agent has registered but that were not added to the master workslip.

- getAgents
public java.util.List getAgents()

- Description
  Returns a list of all agents currently registered with the dispatcher.

- Returns – a list of all known agents.

• getMasterWorkslip

public personalagents.masterworkslip.MasterWorkslip getMasterWorkslip()

- Description
  Returns the master workslip for the dispatcher, which includes all fields needed for the task.

- Returns – the master workslip for the dispatcher.

• registerAgent

public void registerAgent(personalagents.AgentInformation agentInformation)

- Description
  Takes the information from the registering agent and adds it to the list of agents and adds the agent read fields and written fields to the field lookup tables.

- Parameters

  * agentInformation – the information for the new agent
A.8 Package personalagents.masterworkslip

Package Contents

Classes

MasterWorkslip

A list of WorkslipFields that describes all known workslip fields.

A.8.1 Class MasterWorkslip

A list of WorkslipFields that describes all known workslip fields. Update: Version 0.2.0 added getFieldFor method. Update: Version 0.3.0 added addFields method to handle adding multiple fields at once.

Declaration

public class MasterWorkslip
extends java.lang.Object

Constructor summary

MasterWorkslip()

MasterWorkslip(List) Create a list to be populated with all known fields and populate it with the given list of fields.

Method summary

addField(WorkslipField) Adds a field to the master workslip
addFields(Collection) Adds all fields in the collection to the master workslip list of fields.

contains(WorkslipField) Checks that the MasterWorkslip has a value for the given field

findConflictFields(MasterWorkslip) Finds a list of all fields that are contain in both this MasterWorkslip and the argument MasterWorkslip.

getFieldFor(String) Returns a field for a given name.

getFields() Returns all the fields known in this Master Workslip.

removeField(WorkslipField) Removes a field from the master workslip

Constructors

• MasterWorkslip

    public MasterWorkslip()

• MasterWorkslip

    public MasterWorkslip(java.util.List fields)

    – Description
        Create a list to be populated with all known fields and populate it with the given list of fields.

    – Parameters
        * fields – the list of fields to start the list with
Methods

- **addField**

  ```java
  public void addField(personalagents.workslip.WorkslipField field)
  ```

  - **Description**
    
    Adds a field to the master workslip
  
  - **Parameters**
    
    * field - the field to add to the master workslip

- **addFields**

  ```java
  public void addFields(java.util.Collection fields)
  ```

  - **Description**
    
    Adds all fields in the collection to the master workslip list of fields.
  
  - **Parameters**
    
    * fields - the collection of fields to add.

- **contains**

  ```java
  public boolean contains(personalagents.workslip.WorkslipField field)
  ```
A.8. PACKAGE PERSONALAGENTS.MASTERWORKSLIP

- **Description**
  
  Checks that the MasterWorkslip has a value for the given field

- **Parameters**
  
  * field – the field to check

- **Returns** – true if the field is contained in this master workslip

• **findConflictFields**

  ```java
  public java.util.List findConflictFields(MasterWorkslip masterWorkslip)
  ```

  - **Description**
    
    Finds a list of all fields that are contain in both this MasterWorkslip and the argument MasterWorkslip.

  - **Parameters**
    
    * masterWorkslip – the list to compare against this one

  - **Returns** – a list of all fields found in both workslips

• **getFieldFor**

  ```java
  public personalagents.workslip.WorkslipField getFieldFor(java.lang.String fieldName)
  ```

  - **Description**
    
    Returns a field for a given name.
- Parameters
  * fieldName – the name of the field to return

- Returns – the WorkslipField for the given name.

- Throws
  * java.lang.IllegalArgumentException – if the name does not correspond to a field.

• getFields

public java.util.List getFields()

- Description
  Returns all the fields known in this Master Workslip.

- Returns – a copy of the list of all fields

• removeField

public void removeField(personalagents.workslip.WorkslipField field)

- Description
  Removes a field from the master workslip

- Parameters
  * field – the field to remove from the workslip
A.9 Package personalagents.utilities

Classes

RegexMatcher ................................................................. 401
Utility class for matching strings to a regular expression.

WorkslipProducerEngine .................................................. 404
Simple executable class to generate and output workslips.

A.9.1 Class RegexMatcher

Utility class for matching strings to a regular expression.

Declaration

public class RegexMatcher
extends java.lang.Object

Constructor summary

RegexMatcher()

Method summary

matches(Pattern, String) Checks if the candidate string matches the given
    pattern.

returnMatchingArea(Pattern, String) Returns the text in the first matching
    group in the pattern in the candidate string.
returnMatchingAreaForMultiline(Pattern, String) * Returns the text in the first matching group in the pattern in the multi line candidate string.

Constructors

• RegexMatcher

    public RegexMatcher()

Methods

• matches

    public static boolean matches(java.util.regex.Pattern pattern, java.lang.String candidate)

    – Description
        Checks if the candidate string matches the given pattern. If the candidate is null it will throw an IllegalArgumentException.

    – Parameters
        * pattern – the pattern to test the candidate with
        * candidate – the string to test

    – Returns – true if the string matches the pattern

• returnMatchingArea
public static java.lang.String returnMatchingArea(java.util.regex.Pattern pattern, java.lang.String candidate)

- Description

Returns the text in the first matching group in the pattern in the candidate string. If the candidate is null it will throw an IllegalArgumentException.

- Parameters

* pattern – Pattern, including one group to match
* candidate – the string to extract the matching text from

- Returns – the matching text

• returnMatchingAreaForMultiline

public static java.lang.String returnMatchingAreaForMultiline(java.util.regex.Pattern pattern, java.lang.String candidate)

- Description

* Returns the text in the first matching group in the pattern in the multi line candidate string. If the candidate is null it will throw an IllegalArgumentException.

- Parameters

* pattern – Pattern, including one group to match
* candidate – the string to extract the matching text from
A.9.2 Class WorkslipProducerEngine

Simple executable class to generate and output workslips.

Declaration

public class WorkslipProducerEngine
    extends java.lang.Object

Constructor summary

    WorkslipProducerEngine()

Method summary

    main(String[]) Generates an empty workslip in the programs running directory.

Constructors

    • WorkslipProducerEngine

        public WorkslipProducerEngine()

Methods

    • main
public static void main(java.lang.String[] args)

- **Description**
  Generates an empty workslip in the programs running directory.

- **Parameters**
  
  * args –

## A.10 Package personalagents.workslip

### Package Contents

#### Interfaces

- **ObjectDataSerializer** ................................................................. 406

- **WorkslipInterface** ................................................................. 408
  
  Defines the interface for a workslip.

#### Classes

- **DefaultObjectDataSerializer** .................................................. 411
  
  Default implementation of ObjectDataSerializer used during project implementation.

- **Workslip** .................................................................................. 413
  
  Base class for the Task and Workslip workslip.

- **WorkslipField** ................................................................. 422
  
  Define metadata about a field in a workslip.

- **WorkslipSerializer** ................................................................. 427
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

Class to manage writing a workslip as an XML document and creating a workslip from an XML Document

**WorkslipWriter** ................................................................. 429

A.10.1 Interface ObjectDataSerializer

Declaration

```java
public interface ObjectDataSerializer
```

All known subinterfaces

DefaultObjectDataSerializer (in A.10.3, page 411)

All classes known to implement interface

DefaultObjectDataSerializer (in A.10.3, page 411)

Method summary

- `inflateObject(String, Class)` Inflates a given string into an object of class c (which can be determined from a workslip field’s type information).
- `serializeObject(Obieect)` Provides a serialized instance (as a string) of the object.

Methods

- `inflateObject`
java.lang.Object inflateObject(java.lang.String s, java.lang.Class c)

- **Description**
  Inflates a given string into an object of class c (which can be determined from a workslip field’s type information).

- **Parameters**
  * s – the serialized representation of the object
  * c – the class of the object

- **Returns** – the object represented by string s if it is of Class c.

• **serializeObject**

java.lang.String serializeObject(java.lang.Object o)

- **Description**
  Provides a serialized instance (as a string) of the object. If object is an instance of a type known to have a specialized serializer then that serializer is called to return the correct string. If no specialized serializer is known, it returns the objects .toString() result;

- **Parameters**
  * o – the object to be serialized

- **Returns** – the object serialized to its string format
A.10.2 Interface WorkslipInterface

Defines the interface for a workslip. Includes the core functionality of testing if a key is contained in the workslip, getting a raw value from a workslip and reading and writing values from and to the workslip. Update 0.3.0: Replaced String as the key in key/value pairs with new WorkslipField to allow for better management of knowledge about the workslip. Update 0.5.0 Added standardized time format for determining workslip names and new method for getting the workslip name.

See also

- WorkslipField (in A.10.5, page 422)

Declaration

public interface WorkslipInterface

All known subinterfaces

Workslip (in A.10.4, page 413)

All classes known to implement interface

Workslip (in A.10.4, page 413)

Field summary

NAME_TIMESTAMP_FORMAT
Method summary

contains(WorkslipField) Returns true if there is data associated with the given key.

fields()

getValue() Returns the raw object related to the task for this workslip.

getWorkslipName() Returns a unique name for the workslip

read(WorkslipField) Returns the data associated with the given key.

write(WorkslipField, Object) Records data into the workslip.

Fields

• DateTimeFormatter NAME_TIMESTAMP_FORMAT

Methods

• contains

boolean contains(WorkslipField key)

– Description

Returns true if there is data associated with the given key.

– Parameters

* key – the workslip field to check in the workslip

– Returns – true if the field is contained.
• fields

java.util.Collection fields()

• getValue

java.lang.Object getValue()

  – Description
  Returns the raw object related to the task for this workslip.

  – Returns – the ‘raw object’ for the workslip task.

• getWorkslipName

java.lang.String getWorkslipName()

  – Description
  Returns a unique name for the workslip

  – Returns – unique workslip name

• read

java.lang.Object read(WorkslipField key)
-- Description

Returns the data associated with the given key.

-- Parameters

* key – the key to find.

-- Returns

the data or null if the key cannot be found.

• write

void write(WorkslipField key,java.lang.Object data)

-- Description

Records data into the workslip. Checks if the key is associated to the internal prefix.

-- Parameters

* key – the value to record the associate data by.
  * data – the data to record.

A.10.3 Class DefaultObjectDataSerializer

Default implementation of ObjectDataSerializer used during project implementation.

Declaration

public class DefaultObjectDataSerializer
  
  extends java.lang.Object implements ObjectDataSerializer
Constructors

- DefaultObjectDataSerializer

Methods

- inflateObject

public java.lang.Object inflateObject(java.lang.String s, java.lang.Class c)

  - Description
    Returns an object of the given class with values based on the String.

  - Parameters
    * s – the string with value information
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

* c – the class the string is expected to contain information for

– **Returns** – an object of class c loaded with values from string s.

• **serializeObject**

```java
public java.lang.String serializeObject(java.lang.Object o)
```

– **Description**

Produces a string from the given object which can later be restored to an object with the same internal values.

– **Parameters**

* o – the object to be stored

– **Returns** – a string representation of the object.

A.10.4 Class Workslip

Base class for the Task and Workslip workslip. This defines an extensible dictionary that defines data relevant to a given task. An object is held as a raw object that the workslip concerns and all other data is held as key value pairs that can be written and read. Keys are expected to be prefixed with name spaces where necessary. Any data internal to the workslip is prefixed with the workslip internal prefix and this prefix cannot be used by other keys. Classes that extend Workslip should extend the internal prefix. Update 0.3.0: Replaced String as the key in key/value pairs with new WorkslipField to allow for better management of knowledge about the workslip. Update 0.3.1 Added new method to access all fields in this workslip. Update: 0.4.0 Added timestamps for created and updated for both the workslip
as a whole and for each entry. Update: 0.5.0 Added unique workslip name. Update: 0.5.1 Modified setWorkslipName() to allow any renaming

Declaration

public class Workslip
    extends java.lang.Object implements WorkslipInterface

Field summary

    EMPTY_WORKSLIP
    EMPTY_WORKSLIP_NAME

Constructor summary

    Workslip() Creates a new Workslip with no raw data and instantiates a new
dictionary for the key-value pairs.
    Workslip(T) Creates a new Workslip for the given raw data and instantiates a
    new dictionary for the key-value pairs.

Method summary

    checkForErrors(WorkslipField, Object) An empty method which does not
    look for errors in the workslip.
    contains(WorkslipField) Checks if the key is contained (with a value) in this
    workslip.
    fields() Returns a list of all fields in this workslip.
    generateWorkslipName(Workslip) Creates a unique name (based on the
    agent name and timestamp) for the workslip given the name of the current
getCreationTime() Returns the time at which the workslip was created.

getModifedTime() Returns the time at which the last write to the workslip was performed.

getValue() Returns the raw object related to the task for this workslip.

getWorkslipName() Returns a unique name for this workslip based on the agent name and creation time.

getWorkslipPrefix() Returns the key prefix for the workslip.

handleDefinedKeys(WorkslipField, Object) Handles any keys that the workslip should recognize as special.

read(WorkslipField) Returns the data associated with the given key.

setCreationTime(DateTime) Sideloads the creation time to be assigned to serialized workslips.

setModifedTime(DateTime) Sideloads the modification time to be assigned to serialized workslips.

setWorkslipName(String) Sets the workslip name.

write(WorkslipField, Object) Records data into the workslip.

Fields

• public static final Workslip EMPTY_WORKSLIP

• public static final java.lang.String EMPTY_WORKSLIP_NAME

Constructors

• Workslip
public Workslip()

- **Description**
  
  Creates a new Workslip with no raw data and instantiates a new dictionary for the key-value pairs.

- **Workslip**

  public Workslip(java.lang.Object value)

  - **Description**
    
    Creates a new Workslip for the given raw data and instantiates a new dictionary for the key-value pairs.

  - **Parameters**
    
    * value – the raw object related to the task for this workslip.

**Methods**

- **checkForErrors**

  public void checkForErrors(WorkslipField key, java.lang.Object data)

  - **Description**
An empty method which does not look for errors in the workslip. This is implemented to allow the concrete implementation of the `WorkslipInterface` (in A.10.2, page 408) but is expected to be overridden with specific implementations in subclasses.

- **Parameters**
  
  * `key` – the key to check against errors
  * `data` – the value to check with the key

* contains

```java
public boolean contains(WorkslipField key)
```

- **Description**
  
  Checks if the key is contained (with a value) in this workslip.

- **Parameters**
  
  * `key` – the key to check

- **Returns** – true if the key has a value in the workslip.

* fields

```java
public java.util.Collection fields()
```

- **Description**
  
  Returns a list of all fields in this workslip.
• Returns – a list of all fields

• generateWorkslipName

    public static java.lang.String generateWorkslipName(Workslip w)

    – Description

    Creates a unique name (based on the agent name and timestamp) for the workslip
    given the name of the current agent and the workslip creation time.

    – Parameters

    ∗ w – the workslip to name

    – Returns – a new string of the format agentName-CreationTime

• getCreationTime

    public DateTime getCreationTime()

    – Description

    Returns the time at which the workslip was created.

    – Returns – JodaTime DateTime of the creation time.

• getModifiedTime

    public DateTime getModifiedTime()
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

- **Description**
  
  Returns the time at which the last write to the workslip was performed.

- **Returns** – JodaTime DateTime of the write time.

- **getValue**

  ```java
  public java.lang.Object getValue()
  ```

  - **Description**
    
    Returns the raw object related to the task for this workslip.

  - **Returns** – returns the ‘raw value’ for the task the workslip describes.

- **getWorkslipName**

  ```java
  public java.lang.String getWorkslipName()
  ```

  - **Description**
    
    Returns a unique name for this workslip based on the agent name and creation time.

  - **Returns** – a name of the format agentName-CreationTime

- **getWorkslipPrefix**

  ```java
  protected java.lang.String getWorkslipPrefix()
  ```
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

- **Description**

Returns the key prefix for the workslip.

- **Returns** – returns the workslip’s internal prefix.

- **handleDefinedKeys**

  public boolean handleDefinedKeys(WorkslipField key, java.lang.Object data)

- **Description**

Handles any keys that the workslip should recognize as special. In this implementation there are none, but a subclass may override.

- **Parameters**

  * key –

  * data –

- **Returns** –

- **read**

  public java.lang.Object read(WorkslipField key)

- **Description**

Returns the data associated with the given key.

- **Parameters**

  * key – the namespaced key to find.
• **Returns** – the data or null if the key cannot be found.

• **setCreationTime**

protected void setCreationTime(DateTime creationTime)

- **Description**
  
  Sideloading setter to allow serialized workslips to be assigned the correct creation time.

- **Parameters**
  
  * creationTime – JodaTime DateTime of the creation of the workslip

• **setModifiedTime**

protected void setModifiedTime(DateTime modifiedTime)

- **Description**
  
  Sideloading setter to allow serialized workslips to be assigned the correct modification time.

- **Parameters**
  
  * modifiedTime – JodaTime DateTime of the modification of the workslip

• **setWorkslipName**

public void setWorkslipName(java.lang.String name)
- **Description**
  Sets the workslip name

- **Parameters**
  *
  * `name` – the name of the workslip

- **write**

  ```java
  public void write(WorkslipField key, java.lang.Object data)
  ```

  - **Description**
    Records data into the workslip. Checks if the key is associated to the internal prefix.

  - **Parameters**
    *
    * `key` – the namespaced value to record the associate data by.
    *
    * `data` – the data to record.

### A.10.5 Class WorkslipField

Define metadata about a field in a workslip.

**Declaration**

```java
public class WorkslipField
  extends java.lang.Object implements java.lang.Comparable
```
Field summary

**FIELD_SEPARATOR** Symbol used to separate field values in a string

Constructor summary

**WorkslipField(String)** Creates

**WorkslipField(String, Class)** Create a workslip with the given values for name and type, but with the empty string set for potential value and description.

**WorkslipField(String, Class, String, String)** Create a workslip field with the given values.

Method summary

`compareTo(WorkslipField)`

`equals(Object)`

`getDescription()`

`getName()`

`getPotentialValue()`

`getType()`

`hashCode()`

`toString()`

Fields

- public static final java.lang.String **FIELD_SEPARATOR**
  - Symbol used to separate field values in a string
Constructors

- WorkslipField

  public WorkslipField(java.lang.String workslipString) throws java.lang.ClassNotFoundException

  - Description
    Creates
  
  - Parameters
    * workslipString
  
  - Throws
    * java.lang.ClassNotFoundException

- WorkslipField

  public WorkslipField(java.lang.String name, java.lang.Class type)

  - Description
    Create a workslip with the given values for name and type, but with the empty string set for potential value and description.

  - Parameters
    * name – the plain-text name of the field
    * type – the Java class the value of the field should hold
• WorkslipField

public WorkslipField(java.lang.String name, java.lang.Class type, java.lang.String potentialValue, java.lang.String description)

Description
Create a workslip field with the given values.

Parameters
* name – the plain-text name of the field
* type – the Java class the value of the field should hold
* potentialValue – a string describing the allowed values of the field
* description – a human readable description of the field

Methods

• compareTo

public int compareTo(WorkslipField field)

• equals

public boolean equals(java.lang.Object arg0)

• getDescription
public java.lang.String getDescription()

- **Returns** – As a String, a human-readable description of the field

• `getName`

public java.lang.String getName()

- **Returns** – As a String, the canonical name of the workslip field.

• `getPotentialValue`

public java.lang.String getPotentialValue()

- **Returns** – As a String, the range of potential values the value can take

• `getType`

public java.lang.Class getType()

- **Returns** – As a Class, the type the value should be

• `hashCode`

public native int hashCode()
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

- toString

    public java.lang.String toString()

A.10.6 Class WorkslipSerializer

Class to manage writing a workslip as an XML document and creating a workslip from an XML Document

Declaration

    public class WorkslipSerializer
        extends java.lang.Object

Constructor summary

    WorkslipSerializer()

Method summary

    generateWorkslipFileName(String, Workslip)
    inflateWorkslipField(Element)
    serializeWorkslipField(WorkslipField)
    writeToFile(Workslip, String)
    xmlInflate(File)
    xmlSerialize(Workslip)
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

Constructors

- WorkslipSerializer

  public WorkslipSerializer()

Methods

- generateWorkslipFileName

  public static java.lang.String generateWorkslipFileName(java.lang.String
  agentWorkslipFileLocation,Workslip w)

- inflateWorkslipField

  public static WorkslipField inflateWorkslipField(Element e) throws
  java.lang.ClassNotFoundException

- serializeWorkslipField

  public static Element serializeWorkslipField(WorkslipField f)

- writeToFile

  public static boolean writeToFile(Workslip outputWorkslip,java.lang.String
A.10. PACKAGE PERSONALAGENTS.WORKSLIP

workslipFilename)

• xmlInflate

public static Workslip xmlInflate(java.io.File file) throws JDOMException,
javax.xml.stream.XMLStreamException, java.lang.ClassNotFoundException

• xmlSerialize

public static java.lang.String xmlSerialize(Workslip workslip)

A.10.7 Class WorkslipWriter

Declaration

public class WorkslipWriter
extends java.lang.Object

Constructor summary

WorkslipWriter()

Method summary

writeWorkslip(Workslip)
Constructors

- WorkslipWriter

```java
public WorkslipWriter()
```

Methods

- writeWorkslip

```java
public static java.lang.String writeWorkslip(Workslip workslip)
```
Appendix B

System Configuration

This Appendix lists the configurations for all experimental conditions. It includes the configuration for the execution scripts int the pari_values.sh file, and then configurations for the dispatcher and the agents for each conditions.

B.1 An Environmentally Aware Alarm Clock

B.1.1 System $A_1$ for Owen

pari_values.sh

```bash
home_dir=~/data/development/personal_agents/eval/a1/
pari_classpath=lib/*
jar_source=../..../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/
```

431
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive
dispatcher.archive.archivists Light,Full

rwsa.run.config

agent.class replayweathersourceagent.ReplayWeatherSourceAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/weather_source_agent/
agent.workslip.write_location run/input/dispatcher/
source.file.location run/input/weather_source_agent/w7.weather

wea.run.config

agent.class weatheragent.evaluationagent.WeatherEvaluationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_evaluation_agent/
agent.workslip.write_location run/input/dispatcher/

weatherevaluatoragent.evaluator_path_root config/wea/evaluators/

frost.config

evaluator.name Frost

evaluator.class weatheragent.evaluationagent.KeywordEvaluator

evaluator.keyword.read_field weather.source.current.condition # java.lang.String

evaluator.keyword.word Frost
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

evaluator.keyword.positive_score 150
evaluator.keyword.negative_score 0

aca.run.config

agent.class alarmcontrol.AlarmControlAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/alarm_control_agent/
agent.workslip.write_location run/input/dispatcher/

alarmcontrolagent.alarm_path_root config/aca/alarms/

morning.alarm

alarm.name morning
alarm.time 06:00

alarm.fields_tested weather.evaluated.frost.score # java.lang.Float # #
   Evaluation of weather as evaluated for frost

alarm.modificationtable weather.evaluated.frost.score,100,-20
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

asa.run.config

agent.class alarmsounding.agent.AlarmSoundingAgent

agent.execution_model ContinuousRunFile

agent.registration.file_location run/input/dispatcher/

agent.workslip.read_location run/input/alarm_sounding_agent/
agent.workslip.write_location run/input/dispatcher/

alarm.fields alarm.morning.setTime # java.lang.String

B.1.2 System $A_2$ for Tala

pari_values.sh

home_dir="~/data/development/personal_agents/eval/a2/

pari_classpath=lib/*

jar_source=../..///pari/nbproject/dist/PARI.jar

jar_target=lib/PARI.jar

run_data_dir=run/data

agent_input_dir=run/input/

dispatcher.config
agent.class replayweathersourceagent.ReplayWeatherSourceAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_source_agent/
agent.workslip.write_location run/input/dispatcher/

source.file.location run/input/weather_source_agent/w4.weather

rwsa.run.config

agent.class replayweathersourceagent.ReplayWeatherSourceAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_source_agent/
agent.workslip.write_location run/input/dispatcher/

source.file.location run/input/weather_source_agent/w7.weather

wea.run.config
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

agent.class weatheragent.evaluationagent.WeatherEvaluationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_evaluation_agent/
agent.workslip.write_location run/input/dispatcher/

weatherevaluatoragent.evaluator_path_root config/wea/evaluators/

cold.config

evaluator.name Cold

evaluator.class weatheragent.evaluationagent.ValueEvaluator

evaluator.entry weather.source.current.temperature

evaluator.value.table , -20,200#, -20,-15,150#, -15,-10,100

aca.run.config

agent.class alarmcontrol.AlarmControlAgent

agent.execution_model ContinuousRunFile
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

agent.workslip.read_location run/input/alarm_control_agent/

agent.workslip.write_location run/input/dispatcher/

alarmcontrolagent.alarm_path_root config/aca/alarms/

---

**tala.alarm**

---

alarm.name tala

alarm.time 08:00

alarm.fields_tested weather.evaluated.cold.score # java.lang.Float

alarm.modificationtable

    weather.evaluated.cold.score,200,-60#weather.evaluated.cold.score,150,-30#weather.evaluated.cold.score,100,-10

---

**asa.run.config**

---

agent.class alarmsounding.agent.AlarmSoundingAgent

agent.execution_model ContinuousRunFile

agent.registration.file_location run/input/dispatcher/

agent.workslip.read_location run/input/alarm_sounding_agent/
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

agent.workslip.write_location run/input/dispatcher/

alarm.fields alarm.morning.setTime # java.lang.String

B.1.3 Alarm Clock System $A_3$ for Thad

pari_values.sh

home_dir=~/data/development/personal_agents/eval/a3/
pari_classpath=lib/*
jar_source=../..../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output(dispatcher/archive

dispatcher.archive.archivists Light,Full

rwsa.run.config

agent.class replayweathersourceagent.ReplayWeatherSourceAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_source_agent/
agent.workslip.write_location run/input/dispatcher/

source.file.location run/input/weather_source_agent/w6.weather

fpa.run.config

agent.class forecastparsingagent.ForecastParsingAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/forecast_parsing_agent/
agent.workslip.write_location run/input/dispatcher/

parser.source.field weather.source.forecast # java.lang.String
parser.source.keyword snow
parser.source.day 0
parser.source.regex .* (.*) cm.*
parser.produced.field_name weather.parsed.snow

wea1.run.config

agent.class weatheragent.evaluationagent.WeatherEvaluationAgent
agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_evaluation_agent_1/
agent.workslip.write_location run/input/dispatcher/

weatherevaluatoragent.evaluator_path_root config/wea1/evaluators/
snow.config

evaluator.name Snow
evaluator.class weatheragent.evaluationagent.RangeMappingEvaluator
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```java
evaluator.entry weather.parsed.snow # java.lang.Float

evaluator.range.low_value 0
evaluator.range.high_value 20
evaluator.range.low_score 0
evaluator.range.high_score 400
```

```
wea2.run.config
```

```
agent.class weatheragent.evaluationagent.WeatherEvaluationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_evaluation_agent_2/
agent.workslip.write_location run/input/dispatcher/

weatherevaluatoragent.evaluator_path_root config/wea2/evaluators/
```

```
frost.config
```

```
evaluator.name Frost

evaluator.class weatheragent.evaluationagent.KeywordEvaluator
```
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

evaluator.keyword.read_field weather.source.current.condition # java.lang.String

evaluator.keyword.word Frost

evaluator.keyword.positive_score 150
evaluator.keyword.negative_score 0

aca.run.config

agent.class alarmcontrol.AlarmControlAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/alarm_control_agent/
agent.workslip.write_location run/input/dispatcher/

alarmcontrolagent.alarm_path_root config/aca/alarms/

work.alarm

alarm.name work
alarm.time 08:00

alarm.fields_tested weather.evaluated.frost.score #
   java.lang.Float!
weather.evaluated.snow.score # java.lang.Float
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

alarm.modificationtable

weather.evaluated.snow.score,50,-8
weather.evaluated.snow.score,100,-15
weather.evaluated.snow.score,150,-23
weather.evaluated.snow.score,200,-38
weather.evaluated.snow.score,250,-53
weather.evaluated.snow.score,300,-68
weather.evaluated.frost.score,100,-20

asa.run.config

agent.class alarmsounding.agent.AlarmSoundingAgent
agent.execution_model ContinuousRunFile
agent.registration.file_location run/input/dispatcher/
agent.workslip.read_location run/input/alarm_sounding_agent/
agent.workslip.write_location run/input/dispatcher/
alarm.fields alarm.morning.setTime # java.lang.String

B.1.4 Alarm Clock System $A_4$ for Fiona

pari_values.sh

home_dir="/data/development/personal_agents/eval/a4/
pari_classpath=lib/*
jar_source=../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive
dispatcher.archive.archivists Light,Full

rwsa.run.config

agent.class replayweathersourceagent.ReplayWeatherSourceAgent

agent.execution_model ContinuousRunFile
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

agent.workslip.read_location run/input/weather_source_agent/
agent.workslip.write_location run/input/dispatcher/

source.file.location run/input/weather_source_agent/w2.weather

fpa.run.config

agent.class forecastparsingagent.ForecastParsingAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/forecast_parsing_agent/
agent.workslip.write_location run/input/dispatcher/

parser.source.field weather.source.forecast # java.lang.String
parser.source.keyword rain
parser.source.day 0
parser.source.regex .* (.*) mm.*
parser.produced.field_name weather.parsed.rain

wea.run.config

agent.class weatheragent.evaluationagent.WeatherEvaluationAgent
B.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/weather_evaluation_agent/
agent.workslip.write_location run/input/dispatcher/

weatherevaluatoragent.evaluator_path_root config/wea/evaluators/

rain.config

 evaluator.name rain

 evaluator.class weatheragent.evaluationagent.ValueEvaluator

 evaluator.entry weather.parsed.rain # java.lang.Float

 evaluator.value.table 1,25,100#25,,200

aca.run.config

agent.class alarmcontrol.AlarmControlAgent

agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/alarm_control_agent/
agent.workslip.write_location run/input/dispatcher/

alarmcontrolagent.alarm_path_root config/aca/alarms/

bike_time.alarm

alarm.name bike_time
alarm.time 08:30

alarm.fields_tested weather.evaluatd.rain.score # java.lang.Float

alarm.modificationtable
    weather.evaluatd.rain.score,100,-10#weather.evaluatd.rain.score,200,90

asa.run.config

agent.class alarmsounding.agent.AlarmSoundingAgent
agent.execution_model ContinuousRunFile
agent.registration.file_location run/input/dispatcher/
agent.workslip.read_location run/input/alarm_sounding_agent/
agent.workslip.write_location run/input/dispatcher/
B.2 An Efficient and Price Aware Shopping List

B.2.1 System $L_1$ for Olive

pari_values.sh

```
home_dir="~/data/development/personal_agents/eval/l1/
pari_classpath=lib/*
jar_source=../../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/
```

dispatcher.config

```
dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
```
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive

dispatcher.archive.archivists Light,Full

sla.run.config

agent.class shoppinglistagent.ShoppingListAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/shopping_list_agent/

gagent.workslip.write_location run/input/dispatcher/

shoppinglist.file.name run/data/shopping_list.txt

shoppinglist.field_name shopping.list.mylist

sia_a.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_a/
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

agent.workslip.write_location run/input/dispatcher/

store.info.name store_a
store.datafile.name run/data/store/a.txt
store.info.distance 15.0

sia_b.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/sia_b/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_b
store.datafile.name run/data/store/b.txt
store.info.distance 45.0

sia_c.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/sia_c/
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

agent.workslip.write_location run/input/dispatcher/

store.info.name store_c
store.datafile.name run/data/store/c.txt
store.info.distance 1.0

spa.run.config

agent.class shoppingplanagent.ShoppingPlanAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/shopping_plan_agent/
agent.workslip.write_location run/input/dispatcher/
store.names store_a,store_b,store_c
shopping.list.field_name shopping.list.mylist
planner.name distance
plan.distance.penalty .2
plan.mismatch.penalty 4

pca.run.config

agent.class shoppingplanagent.PrincipalCommunicationAgent
agent.execution_model ContinuousRunFile
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

agent.workslip.read_location run/input/principal_communication_agent/
agent.workslip.write_location run/input/dispatcher/

shopping.list.field_name shopping.list.mylist

communication.file.location olive_shopping_plan.txt

B.2.2 System $L_2$ for Taylor

pari_values.sh

```bash
home_dir=/data/development/personal_agents/eval/l2/
pari_classpath=lib/*
jar_source=../../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/
```

dispatcher.config

```bash
dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500
```
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

dispatcher.workslip.input.mode file_reading

dispatcher.workslip.intake.file_location run/input/dispatcher/

dispatcher.workslip.output.mode file_writing

dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive

dispatcher.archive.archivists Light,Full

sla.run.config

agent.class shoppingplanagent.ShoppingPlanAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/shopping_plan_agent/

agent.workslip.write_location run/input/dispatcher/

store.names store_a,store_b,store_c

shopping.list.field_name shopping.list.mylist

planner.name cost

plan.distance.penalty .2

plan.mismatch.penalty 4
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

sia_a.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_a/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_a
store.datafile.name run/data/store/a.txt
store.info.distance 15.0

sia_b.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_b/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_b
store.datafile.name run/data/store/b.txt
store.info.distance 45.0
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

sia_c.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_c/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_c
store.datafile.name run/data/store/c.txt
store.info.distance 1.0

spa.run.config

agent.class shoppingplanagent.ShoppingPlanAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/shopping_plan_agent/
agent.workslip.write_location run/input/dispatcher/

store.names store_a,store_b,store_c
shopping.list.field_name shopping.list.mylist
planner.name cost
plan.distance.penalty .2
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

plan.mismatch.penalty 4

pca.run.config

agent.class shoppingplanagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/principal_communication_agent/
agent.workslip.write_location run/input/dispatcher/

shopping.list.field_name shopping.list.mylist

communication.file.location taylor_shopping_plan.txt

B.2.3 System $L_3$ for Thea

pari_values.sh

home_dir=/data/development/personal_agents/eval/l3/
parsi_classpath=lib/*
jar_source=../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive
dispatcher.archive.archivists Light,Full

sla.run.config

agent.class shoppinglistagent.ShoppingListAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/shopping_list_agent/
agent.workslip.write_location run/input/dispatcher/

shoppinglist.file.name run/data/shopping_list.txt
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

shoppinglist.field_name shopping.list.mylist

sia_a.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_a/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_a
store.datafile.name run/data/store/a.txt
store.info.distance 15.0

sia_b.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/sia_b/
agent.workslip.write_location run/input/dispatcher/

store.info.name store_b
store.datafile.name run/data/store/b.txt
B.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

```plaintext
store.info.distance 45.0

sia_c.run.config

agent.class testshoppinginformationagent.ShoppingInformationAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/sia_c/
agent.workslip.write_location run/input/dispatcher/
store.info.name store_c
store.datafile.name run/data/store/c.txt
store.info.distance 1.0

spa.run.config

agent.class shoppingplanagent.ShoppingPlanAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/shopping_plan_agent/
agent.workslip.write_location run/input/dispatcher/
store.names store_a,store_b,store_c
shopping.list.field_name shopping.list.mylist
```
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

planner.name preference
plan.mismatch.penalty 4

pca.run.config

agent.class shoppingplanagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/principal_communication_agent/
agent.workslip.write_location run/input/dispatcher/

shopping.list.field_name shopping.list.mylist

communication.file.location thea_shopping_plan.txt

B.3 An Informative and Proactive System Operator

B.3.1 System $P_1$ for Oscar

pari_values.sh

home_dir="~/data/development/personal_agents/eval/p1/
pari_classpath=lib/*
jar_source=../pari/nbproject/dist/PARI.jar
jar_target=lib/PARI.jar
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

run_data_dir=run/data
agent_input_dir=run/input/

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive
dispatcher.archive.archivists Light,Full

sta_ssh.config

agent.class noctestagent.NOCTestAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_test_agent/
agent.workslip.write_location run/input/dispatcher/

service.name SSH
service.test.type file
service.test.file.name run/input/noc_test_agent/ssh.status
\begin{lstlisting}
\subsubsection{sta\_http.config}
\begin{lstlisting}
agent.class noctestagent.NOCTestAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/noc_test_agent/
agent.workslip.write_location run/input/dispatcher/

service.name HTTP
service.test.type file
service.test.file.name run/input/noc_test_agent/ssh.status
\begin{lstlisting}
\subsubsection{ssa.config}
\begin{lstlisting}
agent.class nocstatusagent.NOCStatusAgent

\end{lstlisting}
\end{lstlisting}

\end{lstlisting}
\end{lstlisting}

\begin{lstlisting}
\subsubsection{sta\_http.config}
\begin{lstlisting}
agent.class noctestagent.NOCTestAgent
agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/noc_test_agent/
agent.workslip.write_location run/input/dispatcher/

service.name HTTP
service.test.type file
service.test.file.name run/input/noc_test_agent/ssh.status
\begin{lstlisting}
\subsubsection{ssa.config}
\begin{lstlisting}
agent.class nocstatusagent.NOCStatusAgent

\end{lstlisting}
\end{lstlisting}

\end{lstlisting}
\end{lstlisting}

\end{lstlisting}
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_status_agent/
agent.workslip.write_location run/input/dispatcher/

service.names SSH, HTTP
service.status_table.HTTP UP DOWN ULTRA, UP ERROR HIGH, ERROR DOWN ULTRA, DOWN ERROR HIGH, ERROR UP HIGH, DOWN UP HIGH
service.status_table.SSH UP DOWN REGULAR, UP ERROR REGULAR, ERROR DOWN LOW, DOWN ERROR LOW, ERROR UP REGULAR, DOWN UP REGULAR

pco.Ultra.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_ultra_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority ultra
state.log.file.name ultra.log

pco.Ultra.config

agent.class nocstatusagent.PrincipalCommunicationAgent
agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_ultra_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority ultra
state.log.file.name ultra.log

pco_high.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_high_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority high
state.log.file.name high.log

pco_regular.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

agent.workslip.read_location run/input/noc_com_regular_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority regular
state.log.file.name regular.log

pcos_low.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_low_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority low
state.log.file.name low.log

B.3.2 System $P_2$ for Tress

pari_values.sh

home_dir="~/data/development/personal_agents/eval/p2/
pari_classpath=lib/*
jar_source=../../pari/nbproject/dist/PARI.jar
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

jar_target=lib/PARI.jar
run_data_dir=run/data
agent_input_dir=run/input/

dispatcher.config

dispatcher.registration.mode file_reading
dispatcher.registration.file_location run/input/dispatcher/
dispatcher.registration.loop_delay 2500

dispatcher.workslip.input.mode file_reading
dispatcher.workslip.intake.file_location run/input/dispatcher/
dispatcher.workslip.output.mode file_writing
dispatcher.workslip.loop_delay 2500

dispatcher.core.loop_delay 5000

dispatcher.archive.location run/output/dispatcher/archive
dispatcher.archive.archivists Light,Full

sta_ssh.config

agent.class noctestagent.NOCTestAgent

agent.execution_model ContinuousRunFile
agent.workslip.read_location run/input/noc_test_agent/

agent.workslip.write_location run/input/dispatcher/

service.name SSH
service.test.type file
service.test.file.name run/input/noc_test_agent/ssh.status

\begin{lstlisting}
\subsubsection{sta_http.config}
\begin{lstlisting}
agent.class noctestagent.NOCTestAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_test_agent/
agent.workslip.write_location run/input/dispatcher/

service.name HTTP
service.test.type file
service.test.file.name run/input/noc_test_agent/ssh.status

\begin{lstlisting}
\subsubsection{sra_ssh.config}
\begin{lstlisting}
agent.class nocreactionagent.NOCReactionAgent
agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_reaction_ssh_agent/
agent.workslip.write_location run/input/dispatcher/

service.name SSH
service.response.type dummy
service.response.dummy.file_name run/output/nra/ssh.dummy_fixit

\begin{lstlisting}
\subsection{sra\_http.config}
\begin{lstlisting}
agent.class nocreactionagent.NOCReactionAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/nocReaction_http_agent/
agent.workslip.write_location run/input/dispatcher/

service.name SSH
service.response.type dummy
service.response.dummy.file_name run/output/nra/http.dummy_fixit
\end{lstlisting}

\subsubsection{ssa.config}
\begin{lstlisting}
agent.class nocstatusagent.NOCStatusAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_status_agent/
agent.workslip.write_location run/input/dispatcher/

service.names SSH, HTTP

service.status_table.SSH UP DOWN HIGH, UP ERROR HIGH, ERROR DOWN LOW, DOWN ERROR LOW, ERROR UP REGULAR, DOWN UP REGULAR, Alert HIGH

service.status_table.HTTP UP DOWN HIGH, UP ERROR HIGH, ERROR DOWN LOW, DOWN ERROR LOW, ERROR UP REGULAR, DOWN UP REGULAR, Alert HIGH

pc_o_ultra.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_ultra_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority ultra

state.log.file.name ultra.log
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

pcoUltra.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_comUltra_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority ultra
state.log.file.name ultra.log

pcoHigh.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_comHigh_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority high
state.log.file.name high.log

pcoRegular.config
B.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_regular_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority regular
state.log.file.name regular.log

pco_low.config

agent.class nocstatusagent.PrincipalCommunicationAgent

agent.execution_model ContinuousRunFile

agent.workslip.read_location run/input/noc_com_low_agent/
agent.workslip.write_location run/input/dispatcher/

state.priority low
state.log.file.name low.log
Appendix C

(Selected) Workslip Results

This Appendix includes the workslips which show the results of each experimental condition.
This includes each condition and for the Aware Alarm Clock the weather conditions to show
the different actions taken by the agents.

Please Note, in order to save space, only the final workslip for each evaluation is shown
for the examples from Chapter 8. The production of all agents can be determined from the
final workslip produced in the system.

C.1 An Environmentally Aware Alarm Clock

C.1.1 System $A_1$ for Owen

$W_1$

<workslip>
<metaData>
  <name>DEFAULT_EMPTY_WORKSLIP</name>
</metaData>
</workslip>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<creationTime>2018-09-12T11:42:37.635-06:00</creationTime>
<modifiedTime>2018-09-12T11:42:38.397-06:00</modifiedTime>

<workslipEntries>
  <entry>
    <workslipField>
      <name>weather.source.current.humidity</name>
      <type>java.lang.String</type>
      <description>The current humidity</description>
    </workslipField>
    <workslipData>34</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>workslip.agents.processed</name>
      <type>java.lang.String</type>
      <potentialValue>Agent-Names</potentialValue>
      <description>Names all agents that have processed this workslip</description>
    </workslipField>
    <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>weather.source.copyright</name>
    </workslipField>
  </entry>
</workslipEntries>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<type>java.lang.String</type>

<description>The copyright of the creator of the forecast</description>

</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.warnings</name>

<type>java.lang.String</type>

<description>If there are any watches or warnings for the area</description>

</workslipField>

</entry>

<workslipData>

<entry>

<workslipField>

<name>weather.source.forecast</name>

<type>java.lang.String</type>

<description>The forecast for the next several days</description>

</workslipField>

<workslipData>

[ W_ednesday night - Chance of showers. Low plus 1. POP 30 - Partly cloudy. 30 percent chance of showers this evening. Low plus 1., Thursday - Showers. High 9. - Cloudy. Showers beginning near noon. Amount 5 mm. W_ind becoming northwest 20 km/h gusting to 40 in the afternoon. High 9., Thursday night - Flurries. Low minus 2. - Flurries ending overnight then partly cloudy. Amount 2 to 4 cm. W_ind north 20
km/h gusting to 40 becoming light near midnight. Low minus 2., Friday - A mix of sun and cloud. High 6. - A mix of sun and cloud. High 6.,
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</workslipField>

<workslipData>7.1</workslipData>  
</entry>

<entry>

<workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: W_ed Mar 23 23:49:25 MDT 2016</workslipData>  
</entry>

<entry>

<workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
</workslipField>

<workslipData>falling</workslipData>  
</entry>

<entry>

<workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
</workslipField>

<workslipData>24.1 km</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>

<workslipField>
  <name>weather.source.current.observed</name>
  <type>java.lang.String</type>
  <description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.evaluated.frost.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>

<workslipData>0.0</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.current.condition</name>
  <type>java.lang.String</type>
  <description>The current weather condition</description>
</workslipField>

<workslipData>Mostly Cloudy</workslipData>

</entry>
<entry>
    <workslipField>
        <name>weather.source.current.dewpoint</name>
        <type>java.lang.String</type>
        <description>The current dew point</description>
    </workslipField>
    <workslipData>-7.7</workslipData>
</entry>

<entry>
    <workslipField>
        <name>alarm.morning.setTime</name>
        <type>org.joda.time.DateTime</type>
        <description>Alert interested agents that the time for alarm morning should now be the given time</description>
    </workslipField>
    <workslipData>1970-01-01T06:00:00.000-07:00</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.title</name>
        <type>java.lang.String</type>
        <description>The source of the weather</description>
    </workslipField>
    <workslipData>Calgary - W_ether - Environment Canada</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>
<workslipData>SW_ 8 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.environmentCanadaAirQuality</name>
    <type>java.lang.String</type>
    <description>The current air quality as calculated by Environment Canada</description>
  </workslipField>
  <workslipData>3</workslipData>
</entry>
</workslipEntries>
</workslip>

\[ W_2 \]

<workslip>
  <metaData>
    <name>DEFAULT_EMPTY_WORKSLIP</name>
    <creationTime>2018-09-15T17:23:42.641-06:00</creationTime>
    <modifiedTime>2018-09-15T17:23:43.589-06:00</modifiedTime>
  </metaData>
</workslip>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</metaData>
<workslipEntries>
<entry>
<workslipField>
<name>weather.source.current.temperature</name>
<type>java.lang.String</type>
<description>The current temperature</description>
</workslipField>
<workslipData>-0.6</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.visibility</name>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>
<workslipData>8.1 km</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.forecast</name>
<type>java.lang.String</type>
<description>The forecast for the next several days</description>
</workslipField>
<workslipData>[ Thursday night - Flurries or rain showers. Low minus 3. - Cloudy. A few rain showers beginning early this evening then changing
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK


</entry>

<entry>

<workslipField>

<name>weather.source.title</name>

<type>java.lang.String</type>

<description>The source of the weather</description>

</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>
    <workslipField>
        <name>weather.source.current.pressureTrend</name>
        <type>java.lang.String</type>
        <description>The change in barometric pressure</description>
    </workslipField>
    <workslipData>rising</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.current.windChill</name>
        <type>java.lang.String</type>
        <description>The current temperature adjusted for wind chill</description>
    </workslipField>
    <workslipData>-6</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.current.condition</name>
        <type>java.lang.String</type>
        <description>The current weather condition</description>
    </workslipField>
    <workslipData>Mist, Frost</workslipData>
</entry>
<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>

<workslipData>N 23 km/h</workslipData>

<entry>
  <workslipField>
    <name>weather.evaluated.frost.score</name>
    <type>java.lang.Float</type>
    <description>Evaluation of weather as evaluated by Frost Evaluator</description>
  </workslipField>
  <workslipData>150.0</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
  <workslipData>Replay W_eather Source Agent,W_eather Evaluation Agent,Alarm Control Agent</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>
  <workslipField>
    <name>alarm.morning.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm morning should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T05:40:00.000-07:00</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.environmentCanadaAirQuality</name>
    <type>java.lang.String</type>
    <description>The current air quality as calculated by Environment Canada</description>
  </workslipField>
  <workslipData>2</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.humidity</name>
    <type>java.lang.String</type>
    <description>The current humidity</description>
  </workslipField>
  <workslipData>89</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>
  <workslipField>
    <name>weather.source.copyright</name>
    <type>java.lang.String</type>
    <description>The copyright of the creator of the forecast</description>
  </workslipField>
  <workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.recorded_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were recorded</description>
  </workslipField>
  <workslipData>Recorded: 2016-03-25T00:00:05.527-06:00</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Calgary Intl Airport 11</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.dewpoint</name>
  <type>java.lang.String</type>
  <description>The current dew point</description>
</workslipField>

<workslipData>-2.2</workslipData>

<entry>
  <workslipField>
    <name>weather.source.warnings</name>
    <type>java.lang.String</type>
    <description>If there are any watches or warnings for the area</description>
  </workslipField>
  <workslipData />
</entry>

<entry>
  <workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
  </workslipField>
  <workslipData>Published: Thu Mar 24 23:49:22 MDT 2016</workslipData>
</entry>

</workslipEntries>

</workslip>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

\( W_3 \)

\[
<\text{workslip}>

<\text{metaData}>

<\text{name}>DEFAULT\_EMPTY\_W\_ORKSLIP</name>
<\text{creationTime}>2018-09-15T17:40:51.900-06:00</creationTime>
<\text{modifiedTime}>2018-09-15T17:40:53.052-06:00</modifiedTime>
</\text{metaData}>

<\text{workslipEntries}>

<\text{entry}>

<\text{workslipField}>
<\text{name}>weather.source.copyright</name>
<\text{type}>java.lang.String</type>
<\text{description}>The copyright of the creator of the forecast</description>
</\text{workslipField}>
<\text{workslipData}>Copyright 2016, Environment Canada</text{workslipData}>
</\text{entry}>

<\text{entry}>

<\text{workslipField}>
<\text{name}>weather.source.title</name>
<\text{type}>java.lang.String</type>
<\text{description}>The source of the weather</description>
</\text{workslipField}>
<\text{workslipData}>Calgary - Weather - Environment Canada</text{workslipData}>
</\text{entry}>

<\text{entry}>

\]
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>weather.source.current.humidity</name>
  <type>java.lang.String</type>
  <description>The current humidity</description>
</workslipField>

<workslipData>73</workslipData>

<entry>
  <workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
  </workslipField>
  <workslipData>SSW_ 16 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
  <workslipData>Replay W_eather Source Agent, W_eather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
  <workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<entry>
  <workslipField>
    <name>weather.source.current.environmentCanadaAirQuality</name>
    <type>java.lang.String</type>
    <description>The current air quality as calculated by Environment Canada</description>
  </workslipField>
  <workslipData>2</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>24.1 km</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.evaluated.frost.score</name>
    <type>java.lang.Float</type>
    <description>Evaluation of weather as evaluated by Frost Evaluator</description>
  </workslipField>
  <workslipData>0.0</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<entry>
  <workslipField>
    <name>weather.source.current.condition</name>
    <type>java.lang.String</type>
    <description>The current weather condition</description>
  </workslipField>
  <workslipData>Mostly Cloudy</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
  </workslipField>
  <workslipData>rising</workslipData>
</entry>

<entry>
  <workslipField>
    <name>alarm.morning.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm morning should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T06:00:00.000-07:00</workslipData>
</entry>
<workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.temperature</name>
        <type>java.lang.String</type>
        <description>The current temperature</description>
    </workslipField>

    <workslipData>1.8</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.warnings</name>
        <type>java.lang.String</type>
        <description>If there are any watches or warnings for the area</description>
    </workslipField>

    <workslipData />
</entry>

<entry>
    <workslipField>

</workslipField>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<name>weather.source.current.dewpoint</name>
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>
<workslipData>-2.5</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.recorded_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were recorded</description>
</workslipField>
<workslipData>Recorded: 2016-03-26T00:00:06.096-06:00</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.published_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were published</description>
</workslipField>
<workslipData>Published: Fri Mar 25 23:50:22 MDT 2016</workslipData>
</entry>
</workslipEntries>
</workslip>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

\[ W_4 \]

<workslip>

<metaData>

<name>DEFAULT_EMPTY_WORKSLIP</name>

<creationTime>2018-09-15T17:44:59.472-06:00</creationTime>

<modifiedTime>2018-09-15T17:45:01.186-06:00</modifiedTime>

</metaData>

<workslipEntries>

<entry>

<workslipField>

<name>weather.source.copyright</name>

<type>java.lang.String</type>

<description>The copyright of the creator of the forecast</description>

</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.title</name>

<type>java.lang.String</type>

<description>The source of the weather</description>

</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>

</entry>

<entry>

</entry>

<entry>

</entry>

<entry>

</entry>

<entry>
<workslipField>
  <name>weather.source.current.humidity</name>
  <type>java.lang.String</type>
  <description>The current humidity</description>
</workslipField>
<workslipData>52</workslipData>

<entry>
  <workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
  </workslipField>
  <workslipData>S 10 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
  <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<entry>

<workslipField>

<name>weather.source.forecast</name>

<type>java.lang.String</type>

<description>The forecast for the next several days</description>

</workslipField>

<workslipData>[

</entry>

<entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
   <name>weather.source.current.environmentCanadaAirQuality</name>
   <type>java.lang.String</type>
   <description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>2</workslipData>
</entry>

<entry>
   <workslipField>
      <name>weather.evaluated.frost.score</name>
      <type>java.lang.Float</type>
      <description>Evaluation of weather as evaluated by Frost Evaluator</description>
   </workslipField>
   <workslipData>0.0</workslipData>
</entry>

<entry>
   <workslipField>
      <name>weather.source.current.visibility</name>
      <type>java.lang.String</type>
      <description>The current visibility</description>
   </workslipField>
   <workslipData>24.1 km</workslipData>
</entry>
<workslipField>
  <name>weather.source.current.condition</name>
  <type>java.lang.String</type>
  <description>The current weather condition</description>
</workslipField>

<workslipData>Mostly Cloudy</workslipData>

<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>

<workslipData>falling</workslipData>

<workslipField>
  <name>weather.source.current.observed</name>
  <type>java.lang.String</type>
  <description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<entry>
  <workslipField>
    <name>alarm.morning.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm morning should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T06:00:00.000-07:00</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.warnings</name>
    <type>java.lang.String</type>
    <description>If there are any watches or warnings for the area</description>
  </workslipField>
  <workslipData />
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
</entry>
```
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>
<workslipData>-6.4</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.recorded_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were recorded</description>
</workslipField>
<workslipData>Recorded: 2016-03-27T00:00:08.117-06:00</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
</workslipField>
<workslipData>Published: Sat Mar 26 23:49:16 MDT 2016</workslipData>
</entry>

</workslipEntries>
</workslip>

$W_5$
Sunday night - Chance of rain showers or flurries. Low plus 1. POP 60. Partly cloudy. Becoming cloudy this evening with 60 percent chance of rain showers or flurries late this evening and overnight. Wind becoming northwest 20 km/h near midnight. Low plus 1.


Monday night - Periods of snow mixed with rain. Low zero. Periods of snow mixed with rain ending near midnight then partly cloudy. Wind north 20 km/h gusting to 40. Low zero.


Tuesday night - Cloudy periods. Low plus 2.

Wednesday - A mix of sun and cloud. High 11.

Wednesday night - Chance of
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>
<workslipData>-4.3</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.visibility</name>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>
<workslipData>24.1 km</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.humidity</name>
<type>java.lang.String</type>
<description>The current humidity</description>
</workslipField>
<workslipData>54</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>
```
<entry>
<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>
<workslipData>NE 13 km/h</workslipData>
</entry>

<entry>
<workslipField>
  <name>weather.evaluated.frost.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>
<workslipData>150.0</workslipData>
</entry>

<entry>
<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>3</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>

<entry>

<workslipField>
  <name>workslip.agents.processed</name>
  <type>java.lang.String</type>
  <potentialValue>Agent-Names</potentialValue>
  <description>Names all agents that have processed this workslip</description>
</workslipField>

<workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.current.condition</name>
  <type>java.lang.String</type>
  <description>The current weather condition</description>
</workslipField>

<workslipData>Partly Cloudy, Frost</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.warnings</name>
  <type>java.lang.String</type>
  <description>If there are any watches or warnings for the area</description>
</workslipField>

<description>If there are any watches or warnings for the area</description>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>weather.source.recorded_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>Recorded: 2016-03-28T00:00:05.721-06:00</workslipData>

<workslipField>
  <name>alarm.morning.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm morning should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T05:40:00.000-07:00</workslipData>

</workslipEntries>

W_{6}

<workslip>
<description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>3</workslipData>
</entry>
<entry>
<workslipField>
  <name>workslip.agents.processed</name>
  <type>java.lang.String</type>
  <potentialValue>Agent-Names</potentialValue>
  <description>Names all agents that have processed this workslip</description>
</workslipField>
<workslipData>Replay W_eather Source Agent,W_eather Evaluation Agent,Alarm Control Agent</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.evaluated.frost.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>
<workslipData>0.0</workslipData>
</entry>
<entry>
<workslipField>
    <name>weather.source.current.condition</name>
    <type>java.lang.String</type>
    <description>The current weather condition</description>
</workslipField>

<workslipData>Mostly Cloudy</workslipData>

<workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
</workslipField>

<workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

High 16., ]</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.current.visibility</name>
  <type>java.lang.String</type>
  <description>The current visibility</description>
</workslipField>

<workslipData>24.1 km</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.title</name>
  <type>java.lang.String</type>
  <description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - W_ether - Environment Canada</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.current.temperature</name>
  <type>java.lang.String</type>
  <description>The current temperature</description>
</workslipField>

<workslipData>4.6</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>

<workslipField>
  <name>weather.source.published_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were published</description>
</workslipField>
<workslipData>Published: Mon Mar 28 23:48:51 MDT 2016</workslipData>
</entry>
<entry>

<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>
<workslipData>rising</workslipData>
</entry>
<entry>

<workslipField>
  <name>weather.source.current.humidity</name>
  <type>java.lang.String</type>
  <description>The current humidity</description>
</workslipField>
<workslipData>61</workslipData>
</entry>
<entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>alarm.morning.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm morning should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T06:00:00.000-07:00</workslipData>

</entry>

<entry>
  <workslipField>
    <name>weather.source.copyright</name>
    <type>java.lang.String</type>
    <description>The copyright of the creator of the forecast</description>
  </workslipField>

  <workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>

  <workslipData>Calgary Intl Airport 11</workslipData>
</entry>
<name>weather.source.current.dewpoint</name>
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>
<workslipData>-2.3</workslipData>
</entry>

<entry>

<workslipField>
  <name>weather.source.warnings</name>
  <type>java.lang.String</type>
  <description>If there are any watches or warnings for the area</description>
</workslipField>
</entry>
</workslipEntries>
</workslip>

\[ W_7 \]

<workslip>
  <metaData>
    <name>DEFAULT_EMPTY_WORKSLIP</name>
    <creationTime>2018-09-15T17:50:20.932-06:00</creationTime>
    <modifiedTime>2018-09-15T17:50:23.086-06:00</modifiedTime>
  </metaData>
</workslip>
<workslipEntries>
  <entry>
    <workslipField>
      <name>weather.source.current.humidity</name>
      <type>java.lang.String</type>
      <description>The current humidity</description>
    </workslipField>
    <workslipData>36</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>workslip.agents.processed</name>
      <type>java.lang.String</type>
      <potentialValue>Agent-Names</potentialValue>
      <description>Names all agents that have processed this workslip</description>
    </workslipField>
    <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>weather.source.copyright</name>
      <type>java.lang.String</type>
      <description>The copyright of the creator of the forecast</description>
    </workslipField>
  </entry>
</workslipEntries>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipData>Copyright 2016, Environment Canada</workslipData>

<entry>

<workslipField>
  <name>weather.source.warnings</name>
  <type>java.lang.String</type>
  <description>If there are any watches or warnings for the area</description>
</workslipField>

<workslipData />
</entry>

<entry>

<workslipField>
  <name>weather.source.forecast</name>
  <type>java.lang.String</type>
  <description>The forecast for the next several days</description>
</workslipField>

<workslipData>[
  Tuesday night - Increasing cloudiness. Low plus 4. - A few clouds. Increasing cloudiness late this evening. W_ind becoming northwest 20 km/h overnight. Low plus 4.,
  Wednesday - Chance of showers. High 12. POP 30 - Cloudy. 30 percent chance of showers late in the morning and in the afternoon. W_ind northwest 30 km/h gusting to 50. High 12.,
  Wednesday night - Chance of showers. Low plus 2. POP 30 - Cloudy with 30 percent chance of showers in the evening then partly cloudy. W_ind northwest 30 km/h gusting to 50. Low plus 2.,
</workslipData>
Thursday night - Cloudy periods. Low 7. - Cloudy periods. Low 7.,
Friday - Sunny. High 20. - Sunny. High 20., Friday night - Clear. Low
Saturday night - Clear. Low 6. - Clear. Low 6., Sunday - A mix of sun
and cloud. High 14. - A mix of sun and cloud. High 14., Sunday night -
Cloudy periods. Low plus 2. - Cloudy periods. Low plus 2., Monday - A
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>

<workslipData>rising</workslipData>

<entry>
  <workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
  </workslipField>
  <workslipData>Published: Tue Mar 29 23:01:00 MDT 2016</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>24.1 km</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.evaluated.frost.score</name>
  </workslipField>
</entry>
<type>java.lang.Float</type>

<description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>

<workslipData>0.0</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.observed</name>
        <type>java.lang.String</type>
        <description>The location / source of the current data</description>
    </workslipField>
    <workslipData>Calgary Intl Airport 11</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.condition</name>
        <type>java.lang.String</type>
        <description>The current weather condition</description>
    </workslipField>
    <workslipData>Mostly Cloudy</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.dewpoint</name>
        <type>java.lang.String</type>
    </workslipField>
</entry>
<description>The current dew point</description>
</workslipField>

<workslipData>-6.4</workslipData>
</entry>

<entry>

<workslipField>
  <name>alarm.morning.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm morning should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T06:00:00.000-07:00</workslipData>
</entry>

<entry>

<workslipField>
  <name>weather.source.title</name>
  <type>java.lang.String</type>
  <description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>

<entry>

<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
</workslipField>

C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>N/A</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.wind</name>
<type>java.lang.String</type>
<description>The current wind speed</description>
</workslipField>
<workslipData>SSW_ 11 km/h</workslipData>
</entry>
</workslipEntries>
</workslip>

C.1.2 System $A_2$ for Tala

$W_1$

<workslip>
<metaData>

<name>DEFAULT_EMPTY_WORKSLIP</name>
<creationTime>2018-09-25T11:38:57.835-06:00</creationTime>
(modifiedTime>2018-09-25T11:38:59.717-06:00</modifiedTime>
</metaData>
<workslipEntries>
  <entry>
    <workslipField>
      <name>weather.source.current.environmentCanadaAirQuality</name>
      <type>java.lang.String</type>
      <description>The current air quality as calculated by Environment Canada</description>
    </workslipField>
  </entry>
  <workslipData>3</workslipData>
  <entry>
    <workslipField>
      <name>weather.source.current.observed</name>
      <type>java.lang.String</type>
      <description>The location / source of the current data</description>
    </workslipField>
    <workslipData>Calgary Intl Airport 11</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>weather.source.recorded_time</name>
      <type>java.lang.String</type>
      <description>The time at which the conditions were recorded</description>
    </workslipField>
    <workslipData>Recorded: 2016-03-24T00:00:05.829-06:00</workslipData>
  </entry>
</workslipEntries>
C.1. AN ENVIROMENTALLY AWARE ALARM CLOCK

<entry>
  <workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
  </workslipField>
  <workslipData>SW 8 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
  <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>24.1 km</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>

<workslipField>
  <name>weather.source.current.condition</name>
  <type>java.lang.String</type>
  <description>The current weather condition</description>
</workslipField>

<workslipData>Mostly Cloudy</workslipData>
</entry>

<entry>

<workslipField>
  <name>weather.evaluated.cold.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by Cold Evaluator</description>
</workslipField>

<workslipData>0.0</workslipData>
</entry>

<entry>

<workslipField>
  <name>weather.source.title</name>
  <type>java.lang.String</type>
  <description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
  <workslipData>[ Wednesday night - Chance of showers. Low plus 1. POP 30 -
    Partly cloudy. 30 percent chance of showers this evening. Low plus 1.,
    Amount 5 mm. Wind becoming northwest 20 km/h gusting to 40 in the
    afternoon. High 9., Thursday night - Flurries. Low minus 2. - Flurries
    ending overnight then partly cloudy. Amount 2 to 4 cm. Wind north 20
    km/h gusting to 40 becoming light near midnight. Low minus 2., Friday -
    A mix of sun and cloud. High 6. - A mix of sun and cloud. High 6.,
    Friday night - Clear. Low minus 3. - Clear. Low minus 3., Saturday -
    Sunny. High 9. - Sunny. High 9., Saturday night - Clear. Low minus 4. -
    night - Cloudy. Low zero. - Cloudy. Low zero., Monday - Chance of
    flurries or rain showers. High 7. POP 30 - Cloudy with 30 percent
    chance of flurries or rain showers. High 7., Monday night - Chance of
    rain showers or flurries. Low zero. POP 30 - Cloudy periods with 30
    percent chance of rain showers or flurries. Low zero., Tuesday - A mix
    of sun and cloud. High 10. - A mix of sun and cloud. High 10.,
  ]</workslipData>
</entry>

<entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
    <name>weather.source.current.temperature</name>
    <type>java.lang.String</type>
    <description>The current temperature</description>
</workslipField>

<workslipData>7.1</workslipData>

<entry>
    <workslipField>
        <name>alarm.tala.setTime</name>
        <type>org.joda.time.DateTime</type>
        <description>Alert interested agents that the time for alarm tala should now be the given time</description>
    </workslipField>
    <workslipData>1970-01-01T08:00:00.000-07:00</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.published_time</name>
        <type>java.lang.String</type>
        <description>The time at which the conditions were published</description>
    </workslipField>
    <workslipData>Published: Wed Mar 23 23:49:25 MDT 2016</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<entry>
  <workslipField>
    <name>weather.source.current.humidity</name>
    <type>java.lang.String</type>
    <description>The current humidity</description>
  </workslipField>
  <workslipData>34</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
  </workslipField>
  <workslipData>falling</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.copyright</name>
    <type>java.lang.String</type>
    <description>The copyright of the creator of the forecast</description>
  </workslipField>
  <workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
```
<description>The current dew point</description>
 workspaceField
 workspaceData>-7.7</workspaceData>

</entry>

<entry>

 workspaceField
 name>weather.source.warnings</name>
 type>java.lang.String</type>
 description>If there are any watches or warnings for the area</description>
 workspaceField
 workspaceData />

</entry>

</workspaceEntries>

</workspace>

W_2

<workspace>

 metaData>
 name>DEFAULT_EMPTY_WORKSLIP</name>
 creationTime>2018-09-25T11:42:27.368-06:00</creationTime>
 modifiedTime>2018-09-25T11:42:27.978-06:00</modifiedTime>

</metaData>

<workspaceEntries>

 <entry>

<workslipField>
  <name>weather.source.current.humidity</name>
  <type>java.lang.String</type>
  <description>The current humidity</description>
</workslipField>

<workslipData>89</workslipData>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
  <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.copyright</name>
    <type>java.lang.String</type>
    <description>The copyright of the creator of the forecast</description>
  </workslipField>
  <workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<entry>
  <workslipField>
    <name>weather.source.warnings</name>
    <type>java.lang.String</type>
    <description>If there are any watches or warnings for the area</description>
  </workslipField>
  <workslipData />
</entry>

<entry>
  <workslipField>
    <name>alarm.tala.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm tala should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T07:50:00.000-07:00</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
  <workslipData>[
    Thursday night - Flurries or rain showers. Low minus 3. - Cloudy. A few rain showers beginning early this evening then changing

</entry>

<entry>

<workslipField>

<name>weather.evaluated.cold.score</name>

<type>java.lang.Float</type>

<description>Evaluation of weather as evaluated by Cold Evaluator</description>

</workslipField>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipData>100.0</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.recorded_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were recorded</description>
</workslipField>
<workslipData>Recorded: 2016-03-25T00:00:05.527-06:00</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.temperature</name>
  <type>java.lang.String</type>
  <description>The current temperature</description>
</workslipField>
<workslipData>-11.2</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>
<workslipData>rising</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
  </workslipField>
  <workslipData>Published: Thu Mar 24 23:49:22 MDT 2016</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>8.1 km</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Calgary Intl Airport 11</workslipData>
</entry>
<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>
<workslipData>Mist, Frost</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
</workslipField>
<workslipData>-2.2</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.current.windChill</name>
    <type>java.lang.String</type>
    <description>The current temperature adjusted for wind chill</description>
</workslipField>
<workslipData>-6</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.title</name>
    <type>java.lang.String</type>
    <description>The current weather title</description>
</workslipField>
<workslipData></workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.environmentCanadaAirQuality</name>
<type>java.lang.String</type>
<description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>2</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.wind</name>
<type>java.lang.String</type>
<description>The current wind speed</description>
</workslipField>
<workslipData>N 23 km/h</workslipData>
</entry>
</workslipEntries>
</workslip>

\[ W_3 \]
<workslip>
  <metaData>
    <name>DEFAULT_EMPTY_WORKSLIP</name>
    <creationTime>2018-09-25T11:45:03.262-06:00</creationTime>
    <modifiedTime>2018-09-25T11:45:04.357-06:00</modifiedTime>
  </metaData>
  <workslipEntries>
    <entry>
      <workslipField>
        <name>weather.source.current.humidity</name>
        <type>java.lang.String</type>
        <description>The current humidity</description>
      </workslipField>
      <workslipData>73</workslipData>
    </entry>
    <entry>
      <workslipField>
        <name>workslip.agents.processed</name>
        <type>java.lang.String</type>
        <potentialValue>Agent-Names</potentialValue>
        <description>Names all agents that have processed this workslip</description>
      </workslipField>
      <workslipData>Replay Weather Source Agent,Weather Evaluation Agent,Alarm Control Agent</workslipData>
    </entry>
  </workslipEntries>
</workslip>
<entry>
  <workslipField>
    <name>weather.source.copyright</name>
    <type>java.lang.String</type>
    <description>The copyright of the creator of the forecast</description>
  </workslipField>
  <workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.warnings</name>
    <type>java.lang.String</type>
    <description>If there are any watches or warnings for the area</description>
  </workslipField>
  <workslipData />
</entry>

<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
  <workslipData>
    [ Friday night - Chance of showers. Low minus 2. POP 30 - 
      Partly cloudy with 30 percent chance of showers early this evening and risk of a thunderstorm. Clearing late this evening. Wind becoming
  </workslipData>
</entry>

<entry>

<workslipField>

<name>weather.evaluated.cold.score</name>

[type]java.lang.Float[/type]

<description>Evaluation of weather as evaluated by Cold Evaluator</description>

</workslipField>

<workslipData>150.0</workslipData>

</entry>
<entry>

<workslipField>

  <name>weather.source.recorded_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>Recorded: 2016-03-26T00:00:06.096-06:00</workslipData>

</entry>

<entry>

<workslipField>

  <name>weather.source.current.temperature</name>
  <type>java.lang.String</type>
  <description>The current temperature</description>
</workslipField>

<workslipData>-17.5</workslipData>

</entry>

<entry>

<workslipField>

  <name>alarm.tala.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm tala should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T07:30:00.000-07:00</workslipData>

</entry>

<entry>
<workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
</workslipField>
<workslipData>rising</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.published_time</name>
        <type>java.lang.String</type>
        <description>The time at which the conditions were published</description>
    </workslipField>
    <workslipData>Published: Fri Mar 25 23:50:22 MDT 2016</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.current.visibility</name>
        <type>java.lang.String</type>
        <description>The current visibility</description>
    </workslipField>
    <workslipData>24.1 km</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.current.observed</name>
        <type>java.lang.String</type>
        <description>The observed conditions</description>
    </workslipField>
    <workslipData>Observed: Fri Mar 25 23:50:22 MDT 2016</workslipData>
</entry>
<type>java.lang.String</type>
<description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>

<workslipData>Mostly Cloudy</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.current.dewpoint</name>
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>

<workslipData>-2.5</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.title</name>
<type>java.lang.String</type>
<description>The source of the weather</description>
</workslipField>

C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
</workslipField>
<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>2</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>
<workslipData>SSW 16 km/h</workslipData>
</entry>
</workslipEntries>
</workslip>

\[ W_4 \]
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipEntries>
  <entry>
    <workslipField>
      <name>workslip.agents.processed</name>
      <type>java.lang.String</type>
      <potentialValue>Agent-Names</potentialValue>
      <description>Names all agents that have processed this workslip</description>
    </workslipField>
    <workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>weather.source.warnings</name>
      <type>java.lang.String</type>
      <description>If there are any watches or warnings for the area</description>
    </workslipField>
    <workslipData />
  </entry>
</workslipEntries>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<entry>
  <workslipField>
    <name>alarm.tala.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm tala should
       now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T07:00:00.000-07:00</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.condition</name>
    <type>java.lang.String</type>
    <description>The current weather condition</description>
  </workslipField>
  <workslipData>Mostly Cloudy</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.recorded_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were recorded</description>
  </workslipField>
  <workslipData>Recorded: 2016-03-27T00:00:08.117-06:00</workslipData>
</entry>
<workslipField>
    <name>weather.source.current.temperature</name>
    <type>java.lang.String</type>
    <description>The current temperature</description>
</workslipField>
<workslipData>-28.3</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.published_time</name>
        <type>java.lang.String</type>
        <description>The time at which the conditions were published</description>
    </workslipField>
    <workslipData>Published: Sat Mar 26 23:49:16 MDT 2016</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.pressureTrend</name>
        <type>java.lang.String</type>
        <description>The change in barometric pressure</description>
    </workslipField>
    <workslipData>falling</workslipData>
</entry>

<entry>
    <workslipField>
        <name>weather.source.current.visibility</name>
    </workslipField>
</entry>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>

<workslipData>24.1 km</workslipData>

<entry>

<workslipField>

<name>weather.source.forecast</name>
<type>java.lang.String</type>
<description>The forecast for the next several days</description>
</workslipField>

Thursday night - Cloudy periods. Low 7. - Cloudy periods. Low 7.,
Friday - A mix of sun and cloud. High 18. - A mix of sun and cloud.
High 18., ]</workslipData>
</entry>
</entry>
<workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
</workslipField>
<workslipData>-6.4</workslipData>
</entry>
</entry>
<workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
</workslipField>
<workslipData>Calgary Intl Airport 11</workslipData>
</entry>
</entry>
<workslipField>
    <name>weather.source.title</name>
    <type>java.lang.String</type>
    <description>The source of the weather</description>
</workslipField>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>
<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.evaluated.cold.score</name>
<type>java.lang.Float</type>
<description>Evaluation of weather as evaluated by Cold Evaluator</description>
</workslipField>
<workslipData>200.0</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.humidity</name>
<type>java.lang.String</type>
<description>The current humidity</description>
</workslipField>
<workslipData>52</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

C.1.3 System $A_3$ for Thad

$W_1$
<workslip>

<metaData>

  <name>DEFAULT_EMPTY_WORKSLIP</name>

  <creationTime>2018-10-16T11:35:05.501-06:00</creationTime>

  <modifiedTime>2018-10-16T11:35:07.082-06:00</modifiedTime>

</metaData>

<workslipEntries>

  <entry>

    <workslipField>

      <name>weather.source.current.wind</name>

      <type>java.lang.String</type>

      <description>The current wind speed</description>

    </workslipField>

    <workslipData>SW 8 km/h</workslipData>

  </entry>

  <entry>

    <workslipField>

      <name>weather.source.current.condition</name>

      <type>java.lang.String</type>

      <description>The current weather condition</description>

    </workslipField>

    <workslipData>Mostly Cloudy</workslipData>

  </entry>

  <entry>

    <workslipField>

      <name>weather.source.current.environmentCanadaAirQuality</name>

    </workslipField>

</workslipEntries>
<type>java.lang.String</type>
<description>The current air quality as calculated by Environment Canada</description>
</workslipField>
</entry>
<entry>
<workslipField>
<name>weather.source.recorded_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were recorded</description>
</workslipField>
<workslipData>Recorded: 2016-03-24T00:00:05.829-06:00</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.evaluated.frost.score</name>
<type>java.lang.Float</type>
<description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>
<workslipData>0.0</workslipData>
</entry>
<entry>
<workslipField>
<name>a;a.work.setTime</name>
</workslipField>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

[type]org.joda.time.DateTime[/type]
<description>Alert interested agents that the time for alarm work should now be the given time[/description>
</workslipField>
<workslipData>1970-01-01T08:00:00.000-07:00</workslipData>
</entry>
</entry>
<workslipField>
<name>weather.source.warnings</name>
[type]java.lang.String[/type]
<description>If there are any watches or warnings for the area[/description>
</workslipField>
<workslipData />
</entry>
<entry>
<workslipField>
<name>weather.source.current.dewpoint</name>
[type]java.lang.String[/type]
<description>The current dew point[/description>
</workslipField>
<workslipData>-7.7</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.copyright</name>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<type>java.lang.String</type>

<description>The copyright of the creator of the forecast</description>

</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.observed</name>

<type>java.lang.String</type>

<description>The location / source of the current data</description>

</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.pressureTrend</name>

<type>java.lang.String</type>

<description>The change in barometric pressure</description>

</workslipField>

<workslipData>falling</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.title</name>

<type>java.lang.String</type>

<description>The source of the weather</description>

```
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.current.humidity</name>
<type>java.lang.String</type>
<description>The current humidity</description>
</workslipField>
<workslipData>34</workslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.forecast</name>
<type>java.lang.String</type>
<description>The forecast for the next several days</description>
</workslipField>
<workslipData>Wednesday night - 1 cm Snow. Low plus 1. POP 30 - Partly cloudy. 30 percent chance of showers this evening. Low plus 1., Thursday - Showers. High 9. - Cloudy. Showers beginning near noon. Amount 5 mm. Wind becoming northwest 20 km/h gusting to 40 in the afternoon. High 9., Thursday night - Flurries. Low minus 2. - Flurries ending overnight then partly cloudy. Amount 2 to 4 cm. Wind north 20 km/h gusting to 40 becoming light near midnight. Low minus 2., Friday - A mix of sun and cloud. High 6. - A mix of sun and cloud. High 6., Friday night - Clear. Low minus 3. - Clear. Low minus 3., Saturday -</workslipData>
</entry>
<workslipData>1.0</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
  </workslipField>
  <workslipData>Published: Wed Mar 23 23:49:25 MDT 2016</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>24.1 km</workslipData>
</entry>
<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipData>Replay Weather Source Agent, Weather Evaluation Agent, Alarm Control Agent, Forecast Parsing Agent</workslipData>

</entry>
</workslipEntries>
</workslip>

\[ W_2 \]

<workslip>
<metaData>
  <name>DEFAULT_EMPTY_WORKSLIP</name>
  <creationTime>2018-10-16T11:43:24.505-06:00</creationTime>
  <modifiedTime>2018-10-16T11:43:25.146-06:00</modifiedTime>
</metaData>
<workslipEntries>
  <entry>
    <workslipField>
      <name>weather.source.current.humidity</name>
      <type>java.lang.String</type>
      <description>The current humidity</description>
    </workslipField>
    <workslipData>89</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>workslip.agents.processed</name>
    </workslipField>
    <workslipData/>
  </entry>
</workslipEntries>
<type>java.lang.String</type>
<potentialValue>Agent-Names</potentialValue>
<description>Names all agents that have processed this
workslip</description>
</workslipField>

<workslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather
Evaluation Agent, Alarm Control Agent</workslipData>

</entry>
<entry>
<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.warnings</name>
<type>java.lang.String</type>
<description>If there are any watches or warnings for the
area</description>
</workslipField>

<workslipData />
</entry>
<entry>
<workslipField>
  <name>alarm.work.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm work should
  now be the given time</description>
</workslipField>

<workslipData>1970-01-01T07:30:00.000-07:00</workslipData>

<workslipField>
  <name>weather.source.forecast</name>
  <type>java.lang.String</type>
  <description>The forecast for the next several days</description>
</workslipField>

<workslipData>Thursday night - 12 cm Snow. Low minus 3. - Cloudy. A few
rain showers beginning early this evening then changing to flurries
this evening and ending near midnight. Risk of a thunderstorm early
this evening. Wind northeast 20 km/h becoming light this evening. Low
30 percent chance of showers late in the afternoon. High 6., Friday
night - Chance of flurries or rain showers. Low zero. POP 60 - Mainly
cloudy with 60 percent chance of flurries or rain showers in the
evening. Clearing overnight. Low zero., Saturday - Sunny. High 11. -
Sunny. High 11., Saturday night - Clear. Low minus 2. - Clear. Low
minus 2., Sunday - A mix of sun and cloud. High 11. - A mix of sun and
cloud. High 11., Sunday night - Showers. Low zero. - Showers. Low
</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.parsed.snow</name>
<type>java.lang.Float</type>
<description>The value (if any) found in the forecast in field weather.source.forecast day 0 as extracted by the regex .* (.*) cm.*</description>
</workslipField>
<workslipData>12.0</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.recorded_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were recorded</description>
</workslipField>
<workslipData>Recorded: 2016-03-25T00:00:05.527-06:00</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>

<workslipField>
  <name>weather.source.current.temperature</name>
  <type>java.lang.String</type>
  <description>The current temperature</description>
</workslipField>
<workslipData>-0.6</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>
<workslipData>rising</workslipData>

</entry>
<entry>

<workslipField>
  <name>weather.source.published_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were published</description>
</workslipField>
<workslipData>Published: Thu Mar 24 23:49:22 MDT 2016</workslipData>

</entry>
<workslipField>
  <name>weather.source.current.visibility</name>
  <type>java.lang.String</type>
  <description>The current visibility</description>
</workslipField>

<workslipData>8.1 km</workslipData>

<workslipField>
  <name>weather.source.current.observed</name>
  <type>java.lang.String</type>
  <description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>

<workslipField>
  <name>weather.evaluated.frost.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>

<workslipData>0.0</workslipData>
<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>
<workslipData>Mist</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.dewpoint</name>
<type>java.lang.String</type>
<description>The current dew point</description>
</workslipField>
<workslipData>-2.2</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.windChill</name>
<type>java.lang.String</type>
<description>The current temperature adjusted for wind chill</description>
</workslipField>
<workslipData>-6</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.title</name>
<type>java.lang.String</type>
<description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>

<entry>

<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>

<workslipData>2</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>

<workslipData>N 23 km/h</workslipData>

</entry>

</workslipEntries>

</workslip>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslip>

<metaData>

 <name>DEFAULT_EMPTY_WORKSLIP</name>
 <creationTime>2018-10-16T11:48:51.618-06:00</creationTime>
 <modifiedTime>2018-10-16T11:48:52.510-06:00</modifiedTime>
</metaData>

<workslipEntries>

 <entry>

  <workslipField>
   <name>weather.source.warnings</name>
   <type>java.lang.String</type>
   <description>If there are any watches or warnings for the area</description>
  </workslipField>

  <workslipData/>

 </entry>

 <entry>

  <workslipField>
   <name>weather.source.current.humidity</name>
   <type>java.lang.String</type>
   <description>The current humidity</description>
  </workslipField>

  <workslipData>73</workslipData>

 </entry>

 <entry>

  <workslipField>

 </workslipField>

</workslipEntries>
<name>workslip.agents.processed</name>
<type>java.lang.String</type>
<potentialValue>Agent-Names</potentialValue>
<description>Names all agents that have processed this workslip</description>
</workslipField>
<workslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>
<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>
<workslipData>Mostly Cloudy</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>

<workslipData>2</workslipData>

<entry>
  <workslipField>
    <name>weather.source.title</name>
    <type>java.lang.String</type>
    <description>The source of the weather</description>
  </workslipField>
  <workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.parsed.snow</name>
    <type>java.lang.Float</type>
    <description>The value (if any) found in the forecast in field weather.source.forecast day 0 as extracted by the regex .* (.*) cm.*</description>
  </workslipField>
  <workslipData>20.0</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
  </workslipField>
  <workslipData>SSW 16 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
  </workslipField>
  <workslipData>-2.5</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
  </workslipField>
  <workslipData>Published: Fri Mar 25 23:50:22 MDT 2016</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>

  <workslipData>rising</workslipData>
</entry>

  <entry>
    <workslipField>
      <name>weather.source.current.visibility</name>
      <type>java.lang.String</type>
      <description>The current visibility</description>
    </workslipField>

    <workslipData>24.1 km</workslipData>
  </entry>

  <entry>
    <workslipField>
      <name>weather.evaluated.snow.score</name>
      <type>java.lang.Float</type>
      <description>Evaluation of weather as evaluated by Snow Evaluator</description>
    </workslipField>

    <workslipData>400.0</workslipData>
  </entry>

  <entry>
    <workslipField>
      <name>weather.source.current.observed</name>
    </workslipField>
  </entry>

  <entry>
    <workslipField>
      <name>weather.source.current.observed</name>
    </workslipField>
  </entry>

  <entry>
    <workslipField>
      <name>weather.source.current.observed</name>
    </workslipField>
  </entry>

  <entry>
    <workslipField>
      <name>weather.source.current.observed</name>
    </workslipField>
  </entry>
<type>java.lang.String</type>

<description>The location / source of the current data</description>

</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.forecast</name>

<type>java.lang.String</type>

<description>The forecast for the next several days</description>

</workslipField>

<workslipData>


</workslipData>

<entry>

<workslipField>
    <name>alarm.work.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm work should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T07:00:00.000-07:00</workslipData>

</entry>

<entry>

<workslipField>
    <name>weather.source.current.temperature</name>
    <type>java.lang.String</type>
    <description>The current temperature</description>
</workslipField>

<workslipData>1.8</workslipData>

</entry>

<entry>

<workslipField>
    <name>weather.source.recorded_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>1.8</workslipData>

</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

\[ W_4 \]

<workslip>
    <metaData>
        <name>DEFAULT_EMPTY_WORKSLIP</name>
        <creationTime>2018-10-16T11:58:03.759-06:00</creationTime>
        <modifiedTime>2018-10-16T11:58:05.822-06:00</modifiedTime>
    </metaData>
    <workslipEntries>
        <entry>
            <workslipField>
                <name>weather.source.current.humidity</name>
                <type>java.lang.String</type>
                <description>The current humidity</description>
            </workslipField>
            <workslipData>52</workslipData>
        </entry>
        <entry>
            <workslipField>
                <name>workslip.agents.processed</name>
            </workslipField>
        </entry>
    </workslipEntries>
</workslip>

<workslipData>Recorded: 2016-03-26T00:00:06.096-06:00</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<type>java.lang.String</type>
<potentialValue>Agent-Names</potentialValue>
<description>Names all agents that have processed this workslip</description>
</workslipField>

<wrkslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather Evaluation Agent, Alarm Control Agent</wrkslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>

<wrkslipData>Copyright 2016, Environment Canada</wrkslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.warnings</name>
<type>java.lang.String</type>
<description>If there are any watches or warnings for the area</description>
</workslipField>

<wrkslipData />
</entry>

<entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
  <name>weather.source.forecast</name>
  <type>java.lang.String</type>
  <description>The forecast for the next several days</description>
</workslipField>

<workslipData>
<name>weather.source.recorded_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>Recorded: 2016-03-27T00:00:08.117-06:00</workslipData>

<entry>

<workslipField>

<name>weather.source.current.temperature</name>
<type>java.lang.String</type>
<description>The current temperature</description>
</workslipField>

<workslipData>2.5</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.pressureTrend</name>
<type>java.lang.String</type>
<description>The change in barometric pressure</description>
</workslipField>

<workslipData>falling</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.published_time</name>
<type>java.lang.String</type>

</workslipField>


C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: Sat Mar 26 23:49:16 MDT 2016</workslipData>

<entry>

<workslipField>

<name>weather.source.current.visibility</name>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>

<workslipData>24.1 km</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.observed</name>
<type>java.lang.String</type>
<description>The location / source of the current data</description>
</workslipField>

<workslipData>Calgary Intl Airport 11</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>

</entry>
 Mostly Cloudy, Frost

<workslipData>-6.4</workslipData>

1970-01-01T07:40:00.000-07:00

Calgary - Weather - Environment Canada
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

</entry>
<entry>
    <workslipField>
        <name>weather.parsed.snow</name>
        <type>java.lang.Float</type>
        <description>The value (if any) found in the forecast in field weather.source.forecast day 0 as extracted by the regex .* (.*) cm.*</description>
    </workslipField>
    <workslipData>1.0</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.evaluated.snow.score</name>
        <type>java.lang.Float</type>
        <description>Evaluation of weather as evaluated by Snow Evaluator</description>
    </workslipField>
    <workslipData>20.0</workslipData>
</entry>
<entry>
    <workslipField>
        <name>weather.source.current.environmentCanadaAirQuality</name>
        <type>java.lang.String</type>
        <description>The current air quality as calculated by Environment Canada</description>
    </workslipField>
    <workslipData>...</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

\[
W_5
\]

\[
\begin{align*}
\text{weather.source.current.wind} & : \text{java.lang.String} \\
\text{description} & : \text{The current wind speed}
\end{align*}
\]

\[
S \ 10 \ km/h
\]
<type>java.lang.String</type>

<description>The copyright of the creator of the forecast</description>
</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.title</name>
<type>java.lang.String</type>
<description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.current.humidity</name>
<type>java.lang.String</type>
<description>The current humidity</description>
</workslipField>

<workslipData>54</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.current.wind</name>
<type>java.lang.String</type>
<description>The current wind speed</description>
</workslipField>

<workslipData></workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<workslipField>
  <workslipData>NE 13 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.parsed.snow</name>
    <type>java.lang.Float</type>
    <description>The value (if any) found in the forecast in field
    weather.source.forecast day 0 as extracted by the regex .* (.*
    cm.*</description>
  </workslipField>
  <workslipData>12.0</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this
    workslip</description>
  </workslipField>
  <workslipData>Replay Weather Source Agent,Forecast Parsing Agent,Weather
    Evaluation Agent,Alarm Control Agent</workslipData>
</entry>
```
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<name>weather.source.forecast</name>

<type>java.lang.String</type>

<description>The forecast for the next several days</description>

</workslipField>

C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK


<name>weather.source.current.environmentCanadaAirQuality</name>
<type>java.lang.String</type>
<description>The current air quality as calculated by Environment Canada</description>
</workslipField>
<workslipData>3</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.evaluated.frost.score</name>
<type>java.lang.Float</type>
<description>Evaluation of weather as evaluated by Frost Evaluator</description>
</workslipField>
<workslipData>150.0</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.visibility</name>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>
<workslipData>24.1 km</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>
</workslipField>
<workslipData>Partly Cloudy, Frost</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
  </workslipField>
  <workslipData>rising</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Calgary Intl Airport 11</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.temperature</name>
    <type>java.lang.String</type>
    <description>The current temperature</description>
  </workslipField>
  <workslipData>58°F</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<description>The current temperature</description>
</workslipField>
<br>
<workslipData>4.2</workslipData>
</entry>
<br>
<entry>
<workslipField>
  <name>alarm.work.setTime</name>
  <type>org.joda.time.DateTime</type>
  <description>Alert interested agents that the time for alarm work should now be the given time</description>
</workslipField>
<br>
<workslipData>1970-01-01T07:10:00.000-07:00</workslipData>
</entry>
<br>
<entry>
<workslipField>
  <name>weather.source.warnings</name>
  <type>java.lang.String</type>
  <description>If there are any watches or warnings for the area</description>
</workslipField>
<br>
<workslipData />
</entry>
<br>
<entry>
<workslipField>
  <name>weather.source.current.dewpoint</name>
  <type>java.lang.String</type>
</workslipField>

C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<description>The current dew point</description>
</workslipField>

<workslipData>-4.3</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.recorded_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>Recorded: 2016-03-28T00:00:05.721-06:00</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: Sun Mar 27 23:49:13 MDT 2016</workslipData>
</entry>
</workslipEntries>
</workslip>

\[ W_6 \]
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<name>workslip.agents.processed</name>
<type>java.lang.String</type>
<potentialValue>Agent-Names</potentialValue>
<description>Names all agents that have processed this workslip</description>
</workslipField>
<workslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>

<entry>
<workslipField>
<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>
</workslipField>
<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>

<entry>
<workslipField>
<name>weather.source.warnings</name>
<type>java.lang.String</type>
<description>If there are any watches or warnings for the area</description>
</workslipField>
<workslipData />
</entry>
<entry>
  <workslipField>
    <name>alarm.work.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm work should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T06:40:00.000-07:00</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
  Saturday night - Clear. Low plus 5. - Clear. Low plus 5., Sunday - A
  mix of sun and cloud. High 16. - A mix of sun and cloud. High 16.,
</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<type>java.lang.String</type>
<description>The current temperature</description>
</workslipField>

<workslipData>4.6</workslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.current.pressureTrend</name>
<type>java.lang.String</type>
<description>The change in barometric pressure</description>
</workslipField>

<workslipData>rising</workslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.published_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: Mon Mar 28 23:48:51 MDT 2016</workslipData>
</entry>

<entry>

<workslipField>
<name>weather.source.current.visibility</name>
<type>java.lang.String</type>
<description>The current visibility</description>
</workslipField>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Calgary Intl Airport 11</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.condition</name>
    <type>java.lang.String</type>
    <description>The current weather condition</description>
  </workslipField>
  <workslipData>Mostly Cloudy, Frost</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
  </workslipField>
  <workslipData>-2.3</workslipData>
</entry>
```
</entry>

<entry>

<workslipField>
  <name>weather.source.title</name>
  <type>java.lang.String</type>
  <description>The source of the weather</description>
</workslipField>

<workslipData>Calgary - Weather - Environment Canada</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>

<workslipData>3</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.current.wind</name>
  <type>java.lang.String</type>
  <description>The current wind speed</description>
</workslipField>

<workslipData>NNW 19 km/h</workslipData>

</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

C.1.4 System $A_4$ for Fiona

$W_2$

```xml
<workslip>
  <metaData>
    <name>DEFAULT_EMPTY_WORKSLIP</name>
    <creationTime>2019-02-12T17:18:21.062-07:00</creationTime>
    <modifiedTime>2019-02-12T17:18:22.196-07:00</modifiedTime>
  </metaData>
  <workslipEntries>
    <entry>
      <workslipField>
        <name>weather.source.current.humidity</name>
        <type>java.lang.String</type>
        <description>The current humidity</description>
      </workslipField>
      <workslipData>88</workslipData>
    </entry>
    <entry>
      <workslipField>
        <name>workslip.agents.processed</name>
        <type>java.lang.String</type>
      </workslipField>
    </entry>
  </workslipEntries>
</workslip>
```
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<potentiaValue>Agent-Names</potentiaValue>
<description>Names all agents that have processed this workslip</description>
</workslipField>
<workslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.copyright</name>
  <type>java.lang.String</type>
  <description>The copyright of the creator of the forecast</description>
</workslipField>
<workslipData>Copyright 2016, Environment Canada</workslipData>
</entry>
<entry>
<workslipField>
  <name>weather.source.warnings</name>
  <type>java.lang.String</type>
  <description>If there are any watches or warnings for the area</description>
</workslipField>
<workslipData />
</entry>
<entry>
<workslipField>
</workslipField>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<name>weather.source.forecast</name>
<type>java.lang.String</type>
<description>The forecast for the next several days</description>
</workslipField>

<workslipData>
</workslipData>

<entry>
<workslipField>
<name>weather.parsed.rain</name>
<type>java.lang.Float</type>
<description>The value (if any) found in the forecast in field weather.source.forecast day 0 as extracted by the regex .* (.* \(.*\)) mm.*</description>
</workslipField>
<workslipData>5.0</workslipData>
</entry>
<entry>
<workslipField>
<name>weather.source.recorded_time</name>
<type>java.lang.String</type>
<description>The time at which the conditions were recorded</description>
</workslipField>

<workslipData>Recorded: 2016-03-25T00:00:05.527-06:00</workslipData>

<entry>

<workslipField>
  <name>weather.source.current.temperature</name>
  <type>java.lang.String</type>
  <description>The current temperature</description>
</workslipField>

<workslipData>-3</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.published_time</name>
  <type>java.lang.String</type>
  <description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: Thu Mar 24 23:49:22 MDT 2016</workslipData>

</entry>

<entry>

<workslipField>
  <name>weather.source.current.pressureTrend</name>
  <type>java.lang.String</type>
  <description>The change in barometric pressure</description>
</workslipField>

</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipData>dropping</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
  </workslipField>
  <workslipData>8.1 km</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Ottawa Intl Airport 11</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
  </workslipField>
  <workslipData>-2.2</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.condition</name>
    <type>java.lang.String</type>
    <description>The current weather condition</description>
  </workslipField>
  <workslipData>Rain 5mm</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.windChill</name>
    <type>java.lang.String</type>
    <description>The current temperature adjusted for wind chill</description>
  </workslipField>
  <workslipData>-9</workslipData>
</entry>

<entry>
  <workslipField>
    <name>alarm.bike_time.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm bike_time should now be the given time</description>
  </workslipField>
  <workslipData>1970-01-01T08:00:00.000-07:00</workslipData>
</entry>
<workslipField>
  <name>weather.source.title</name>
  <type>java.lang.String</type>
  <description>The source of the weather</description>
</workslipField>

<workslipData>Ottawa - Weather - Environment Canada</workslipData>

<workslipField>
  <name>weather.evaluated.rain.score</name>
  <type>java.lang.Float</type>
  <description>Evaluation of weather as evaluated by rain Evaluator</description>
</workslipField>

<workslipData>100.0</workslipData>

<workslipField>
  <name>weather.source.current.environmentCanadaAirQuality</name>
  <type>java.lang.String</type>
  <description>The current air quality as calculated by Environment Canada</description>
</workslipField>

<workslipData>2</workslipData>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
</workslipField>

<workslipData>E 23 km/h</workslipData>

<workslipEntries>
</workslip>

W_3

<workslip>
    <metaData>
        <name>DEFAULT_EMPTY_WORKSLIP</name>
        <creationTime>2019-02-12T17:34:24.760-07:00</creationTime>
        <modifiedTime>2019-02-12T17:34:26.264-07:00</modifiedTime>
    </metaData>
    <workslipEntries>
        <entry>
            <workslipField>
                <name>weather.source.warnings</name>
                <type>java.lang.String</type>
                <description>If there are any watches or warnings for the area</description>
            </workslipField>
        </entry>
    </workslipEntries>
</workslip>
<workslipData />
</entry>

<entry>
  <workslipField>
    <name>weather.evaluated.rain.score</name>
    <type>java.lang.Float</type>
    <description>Evaluation of weather as evaluated by rain Evaluator</description>
  </workslipField>
  <workslipData>200.0</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.humidity</name>
    <type>java.lang.String</type>
    <description>The current humidity</description>
  </workslipField>
  <workslipData>88</workslipData>
</entry>

<entry>
  <workslipField>
    <name>workslip.agents.processed</name>
    <type>java.lang.String</type>
    <potentialValue>Agent-Names</potentialValue>
    <description>Names all agents that have processed this workslip</description>
  </workslipField>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

<workslipField>

<workslipData>Replay Weather Source Agent, Forecast Parsing Agent, Weather Evaluation Agent, Alarm Control Agent</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.copyright</name>
<type>java.lang.String</type>
<description>The copyright of the creator of the forecast</description>

</workslipField>

<workslipData>Copyright 2016, Environment Canada</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.source.current.condition</name>
<type>java.lang.String</type>
<description>The current weather condition</description>

</workslipField>

<workslipData>Rain 5mm</workslipData>

</entry>

<entry>

<workslipField>

<name>weather.parsed.rain</name>
<type>java.lang.Float</type>
<description>The value (if any) found in the forecast in field weather.source.forecast day 0 as extracted by the regex .* (.*)
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

mm.*</description>
</workslipField>

<workslipData>26.0</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.title</name>
    <type>java.lang.String</type>
    <description>The source of the weather</description>
</workslipField>

<workslipData>Ottawa - Weather - Environment Canada</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.current.environmentCanadaAirQuality</name>
    <type>java.lang.String</type>
    <description>The current air quality as calculated by Environment Canada</description>
</workslipField>

<workslipData>2</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.current.windChill</name>
    <type>java.lang.String</type>
    <description>The current temperature adjusted for wind chill</description>
</workslipField>

<workslipData>2</workslipData>
</entry>
C.1. AN ENVIRONMENTALLY AWARE ALARM CLOCK

```xml
<workslipField>
  <workslipData>-9</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.wind</name>
    <type>java.lang.String</type>
    <description>The current wind speed</description>
  </workslipField>
  <workslipData>E 23 km/h</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.dewpoint</name>
    <type>java.lang.String</type>
    <description>The current dew point</description>
  </workslipField>
  <workslipData>-2.2</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.pressureTrend</name>
    <type>java.lang.String</type>
    <description>The change in barometric pressure</description>
  </workslipField>
  <workslipData>dropping</workslipData>
</entry>
```
<entry>

<workslipField>
    <name>weather.source.published_time</name>
    <type>java.lang.String</type>
    <description>The time at which the conditions were published</description>
</workslipField>

<workslipData>Published: Thu Mar 24 23:49:22 MDT 2016</workslipData>
</entry>

<entry>

<workslipField>
    <name>weather.source.current.visibility</name>
    <type>java.lang.String</type>
    <description>The current visibility</description>
</workslipField>

<workslipData>8.1 km</workslipData>
</entry>

<entry>

<workslipField>
    <name>alarm.bike_time.setTime</name>
    <type>org.joda.time.DateTime</type>
    <description>Alert interested agents that the time for alarm bike_time should now be the given time</description>
</workslipField>

<workslipData>1970-01-01T10:00:00.000-07:00</workslipData>
</entry>
<entry>
  <workslipField>
    <name>weather.source.current.observed</name>
    <type>java.lang.String</type>
    <description>The location / source of the current data</description>
  </workslipField>
  <workslipData>Ottawa Intl Airport 11</workslipData>
</entry>

<entry>
  <workslipField>
    <name>weather.source.forecast</name>
    <type>java.lang.String</type>
    <description>The forecast for the next several days</description>
  </workslipField>
</entry>

<entry>
  <workslipField>
    <name>weather.source.current.temperature</name>
    <type>java.lang.String</type>
  </workslipField>
</entry>
C.2 An Efficient and Price Aware Shopping List

C.2.1 System $L_1$ for Olive

<workslip>
  <metaData>
    <name>Shopping Plan Agent_Workslip</name>
    <creationTime>2019-02-17T20:11:35.592-07:00</creationTime>
    <modifiedTime>2019-02-17T20:11:37.817-07:00</modifiedTime>
  </metaData>
</workslip>
<workslipEntries>
  <entry>
    <workslipField>
      <name>store.store_c.product</name>
      <type>java.lang.String</type>
      <description>The product information for store_c, as commaseparated
       triples of item, brand, price. Eachtriple is seperated by a #
       symbol.</description>
    </workslipField>
    <workslipData>1 Milk, Palm, 12.49#Strawberry Jam, Smuckers, 4.00#Olive Oil,
       Oliveri, 8.48#Olive Oil, Store C, 5.48#Bananas, Tree,
       1.79#</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>store.store_c.distance</name>
      <type>java.lang.Float</type>
      <description>The distance to store_c stored as a travel time (in
       minutes).</description>
    </workslipField>
    <workslipData>1.0</workslipData>
  </entry>
  <entry>
    <workslipField>
      <name>workslip.agents.processed</name>
      <type>java.lang.String</type>
    </workslipField>
  </entry>
</workslipEntries>
C.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

C.2.2 System $L_2$ for Taylor

<workslip>
<metaData>
  <name>Shopping Plan Agent_Workslip</name>
  <creationTime>2019-02-17T20:17:42.328-07:00</creationTime>
</metaData>
</workslip>
C.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<modifiedTime>2019-02-17T20:17:44.558-07:00</modifiedTime>
</metaData>
<workslipEntries>
<entry>
<workslipField>
  <name>workslip.agents.processed</name>
  <type>java.lang.String</type>
  <potentialValue>Agent-Names</potentialValue>
  <description>Names all agents that have processed this
  workslip</description>
</workslipField>
<workslipData>Store Information Agent, Shopping Plan Agent</workslipData>
</entry>
<entry>
<workslipField>
  <name>store.store_c.distance</name>
  <type>java.lang.Float</type>
  <description>The distance to store_c stored as a travel time (in
  minutes).</description>
</workslipField>
<workslipData>1.0</workslipData>
</entry>
<entry>
<workslipField>
  <name>store.store_c.product</name>
  <type>java.lang.String</type>
<description>The product information for store_c, as commaseparated
triples of item, brand, price. Eachtriple is seperated by a #
symbol.</description>
</workslipField>
<workslipData>1 Milk, Palm, 12.49#Strawberry Jam, Smuckers, 4.00#Olive Oil,
Oliveri, 8.48#Olive Oil, Store C, 5.48#Bananas, Tree,
1.79#</workslipData>
</entry>
<entry>
<workslipField>
<name>shopping.plan.cost</name>
<type>java.lang.String</type>
<description>The plan for shopping according to the cost
planner</description>
</workslipField>
<workslipData>store_a: Kraft-Strawberry Jam, store_b: Palm-1 Milk Store
B-Olive Oil Tree-Bananas, No Store Carrying: -Oranges</workslipData>
</entry>
</workslipEntries>
</workslip>

C.2.3 System $L_3$ for Thea

<workslip>
<metaData>
<name>Shopping Plan Agent_Workslip</name>
C.2. AN EFFICIENT AND PRICE AWARE SHOPPING LIST

<creationTime>2019-02-17T21:38:925-07:00</creationTime>
<modifiedTime>2019-02-17T21:39:924-07:00</modifiedTime>
</metaData>

<workslipEntries>
<entry>
<workslipField>
  <name>workslip.agents.processed</name>
  <type>java.lang.String</type>
  <potentialValue>Agent-Names</potentialValue>
  <description>Names all agents that have processed this workslip</description>
</workslipField>

<workslipData>Store Information Agent,Shopping Plan Agent</workslipData>
</entry>

<entry>
<workslipField>
  <name>store.store_c.product</name>
  <type>java.lang.String</type>
  <description>The product information for store_c, as comma separated triples of item, brand, price. Each triple is separated by a # symbol.</description>
</workslipField>

<workslipData>1 Milk, Palm, 12.49#Strawberry Jam, Smuckers, 4.00#Olive Oil, Oliveri, 8.48#Olive Oil, Store C, 5.48#Bananas, Tree, 1.79#</workslipData>
</entry>
</workslipEntries>
C.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

C.3.1 System $P_1$ for Oscar
<workslip>
  <metaData>
    <name>NOC Status Agent_Workslip</name>
    <creationTime>2019-02-19T23:19:49.950-07:00</creationTime>
    <modifiedTime>2019-02-20T14:50:07.177-07:00</modifiedTime>
  </metaData>
  <workslipEntries>
    <entry>
      <workslipField>
        <name>noc.state.update</name>
        <type>java.lang.String</type>
        <potentialValue>&lt;service&gt;: &lt;update&gt;</potentialValue>
        <description>An update to the state of the services</description>
      </workslipField>
      <workslipData>SSH: RESTORED</workslipData>
    </entry>
    <entry>
      <workslipField>
        <name>noc.state.detail</name>
        <type>java.lang.String</type>
        <description>The detail of the state information</description>
      </workslipField>
      <workslipData>SSH has error of 2</workslipData>
    </entry>
    <entry>
      ...
    </entry>
  </workslipEntries>
</workslip>
<workslipField>
  <name>workslip.agents.processed</name>
  <type>java.lang.String</type>
  <potentialValue>Agent-Names</potentialValue>
  <description>Names all agents that have processed this workslip</description>
</workslipField>

<workslipData>NOC Test Agent, NOC Status Agent</workslipData>

<entry>
  <workslipField>
    <name>noc.SSH.status</name>
    <type>java.lang.String</type>
    <potentialValue>Up, Down, Error</potentialValue>
    <description>Current Status of service SSH</description>
  </workslipField>
  <workslipData>UP</workslipData>
</entry>

<entry>
  <workslipField>
    <name>noc.state.priority</name>
    <type>java.lang.String</type>
    <potentialValue>Low, Regular, High, Ultra</potentialValue>
    <description>The priority of the state information</description>
  </workslipField>
  <workslipData>HIGH</workslipData>
</entry>
C.3. AN INFORMATIVE AND PROACTIVE SYSTEM OPERATOR

C.3.2 System $P_2$ for Tress

```xml
<workslip>
  <metaData>
    <name>NOC Status Agent_Workslip</name>
    <creationTime>2019-02-20T16:53:09.055-07:00</creationTime>
    <modifiedTime>2019-03-23T17:14:46.667-06:00</modifiedTime>
  </metaData>
  <workslipEntries>
    <entry>
      <workslipField>
        <name>noc.SSH.reaction.detail</name>
        <type>java.lang.String</type>
        <description>Details of an attempt to restore SSH</description>
      </workslipField>
      <workslipData>Reaction Alert reacted to noc.SSH.status contains ERROR and succeeded</workslipData>
    </entry>
    <entry>
      <workslipField>
        <name>noc.state.priority</name>
        <type>java.lang.String</type>
      </workslipField>
    </entry>
  </workslipEntries>
</workslip>
```
<potentialValue>Low, Regular, High, Ultra</potentialValue>
<description>The priority of the state information</description>
</workslipField>
<workslipData>HIGH</workslipData>
</entry>
<entry>
<workslipField>
<name>noc.SSH.error</name>
<type>java.lang.Integer</type>
<description>Error code of service SSH if error status</description>
</workslipField>
<workslipData>2</workslipData>
</entry>
<entry>
<workslipField>
<name>noc.state.update</name>
<type>java.lang.String</type>
<potentialValue>&lt;service&gt;: &lt;update&gt;</potentialValue>
<description>An update to the state of the services</description>
</workslipField>
<workslipData>SSH: DOWN</workslipData>
</entry>
<entry>
<workslipField>
<name>workslip.agents.processed</name>
<type>java.lang.String</type>
<potentialValue>Agent-Names</potentialValue>

<description>Names all agents that have processed this workslip</description>
</workslipField>

<workslipData>NOC Test Agent, NOC Reaction Agent, NOC Status Agent</workslipData>
</entry>

<entry>

<workslipField>

<name>noc.SSH.reaction.status</name>

?type>java.lang.String</type>

<description>Status of an attempt to restore SSH</description>
</workslipField>

<workslipData>Alert Successful Attempt</workslipData>
</entry>

<entry>

<workslipField>

<name>noc.SSH.status</name>

?type>java.lang.String</type>

<potentialValue>Up, Down, Error</potentialValue>

<description>Current Status of service SSH</description>
</workslipField>

<workslipData>ERROR</workslipData>
</entry>
<name>noc.state.detail</name>
<type>java.lang.String</type>
<description>The detail of the state information</description>
</workslipField>
<workslipData>
SSH reaction Alert Successful Attempt: Reaction Alert reacted to noc.SSH.status contains ERROR and succeeded
</workslipData>
</entry>
</workslipEntries>
</workslip>