The End-to-End Use of Source Code Examples: An Exploratory Study—Appendix

Reid Holmes
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA, USA
rtholmes@cs.washington.edu

Rylan Cottrell, Robert J. Walker, Jörg Denzinger
Dept. of Computer Science
University of Calgary
Calgary, AB, Canada
cottrell, rwalker, denzinge@cpsc.ucalgary.ca

Technical report 2009-934-13

Abstract

This appendix contains the details of our case studies outlined in our paper for the 2009 International Conference on Software Maintenance*, as well as an expanded discussion section. The reader is directed to the main paper for introduction, motivation, and related work.*

A. Case Studies

We conducted case studies into the location, selection, and integration of source code examples in the context of particular tasks. Four scenarios were selected from the published literature on source code examples. The scenarios were chosen to possess some complexity and to display some variety in terms of their domain and attributes.

The method and criteria that we applied for this process are described in Section A.1. Section A.2 describes the scenarios, their associated code skeletons and queries, and overviews the results from each scenario. Section A.3 analyzes the results and describes a few general observations.

A.1. Method

For each scenario, we constructed a code skeleton—based on the original article's description—to represent the point at which the developer required assistance from an example.

Locating and selecting examples. GCS requires a string-based query, which we manually extracted from the code skeleton; we queried the web site with this and examined the returned examples. To investigate an example, we selected the source file and visually scanned through it, concentrating our analysis on the GCS-highlighted parts of the file. To query Strathcona,[1] we highlighted the relevant source code in our code skeleton and initiated a query. We examined the returned graphical examples, investigating the highlighted source code for examples that looked promising.

We analyzed the first five returned examples from each location approach and evaluated their relevance to the scenario, in order. We limited this investigation to five examples, as some empirical evidence indicates that developers rarely look beyond this limit when searching [9].

Our main goal when investigating a search result was to determine whether the example provided the functionality required by the task. To do this we visually scanned all the highlighted regions returned by GCS along with the selection provided by Strathcona. If the example seemed functionally relevant, we marked it as a candidate for integration. We also noted the number of methods that were highlighted by the search approach to gain a sense for how much code we had to scan through with each search technique as well as the time we required to make our relevance determination. Table 1 provides an overview of the search process for the four scenarios we investigated.

In a real development situation, a developer is likely to halt the investigation upon discovery of a method that seems promising; investigation might resume if the results of the attempted integration were less than satisfactory. Despite this and for the sake of completeness, we report the time and results for investigating each of the five examples for both approaches.

Integrating potentially relevant examples. For each example deemed relevant to the task using either location approach, we attempted to integrate the example into our code skeleton. For the integration phase, we independently used the standard tools available in the Eclipse IDE ("manual" approach) and the Jigsaw tool [2].

Manual integration involved copying the relevant code

[1] The repository contained all of Eclipse 3.3 and NetBeans 6.

from the example, pasting it into our code skeleton, and modifying it to resolve compilation errors and so it would work in our environment. Jigsaw integration involved selecting the originating source method that the developer wanted to reuse and selecting the target method in the code skeleton; this activated Jigsaw which subsequently copied the required code to our system and modified it to compile within our system. Before Jigsaw could be queried in this way, we had to setup an environment for it. This involved copying the full source file to our system and making it fully compile; sometimes this would mean modifying the code to remove those portions that were not compiling and were irrelevant to our class; other times this meant importing more code to enable the example code to compile. This compilation limitation arises because of requirements Jigsaw has on the Eclipse AST; we discuss this issue further in Section A.3.

To quantitatively compare the integration approaches, we record the time required to perform the task (and the time required to setup Jigsaw, where applicable), the lines of code (LOC) ultimately reused from the example, and the number of discrete actions required of the developer to perform the task. These results are found in Table 2.

We use "actions" in this context as an effort indicator that is alternative to time. We counted any specific decision a developer made as an action; for example, if the developer decided to remove a certain method, deleting the method and all calls to it counted as a single action. Copying a method and resolving any dangling dependencies counted as a single action. Any unique modification to the source code counted as an action.

A.2. Scenarios

Four scenarios were used in our case studies, one each from the publications on Strathcona [7], XSnippet [8], PARSEWeb [10], and XFinder [4]. We examine these, in turn, below. As a detailed description of these integration tasks would be overwhelming for the reader, we opted to provide full details for the first scenario while providing only the most significant, qualitative details for the rest.

Scenario 1: Compute the signature of a `MethodDeclaration`. The Eclipse abstract syntax tree (AST) maintains a programmatic tree-based representation of the source code that the environment has compiled. These ASTs represent Java methods using `MethodDeclaration` objects. This scenario involves the generation of method signatures (e.g., `Object getResults(int[], Vector)`) from `MethodDeclaration` objects. The code skeleton for the task is given in Figure 1 (from Holmes et al. [7, Figure 6d]). This skeleton is motivated by the fact that the developer can quickly discover these methods as apparently relevant to their task, but remain unsure of how to manipulate them appropriately. For the Strathcona query, we selected lines 7 to 9 of the skeleton; we queried GCS with "`ASTVisitor MethodDeclaration getModifiers getName getIdentifier parameters lang:Java`".

```
 1  import java.util.List;
 2  import org.eclipse.jdt.core.dom.ASTVisitor;
 3  import org.eclipse.jdt.core.dom.
        MethodDeclaration;
 4
 5  public class ExtractSig extends ASTVisitor {
 6    public boolean visit(MethodDeclaration node)
          {
 7      int mod = node.getModifiers();
 8      String id = node.getName().getIdentifier();
 9      List parms = node.parameters();
10
11      return super.visit(node);
12    }
13  }
```

Figure 1. Scenario 1 code skeleton.

Location and selection. GCS returned 102 examples. Of the first five, three of them matched all the queried terms (G12, G13, G15). The first and fourth examples (G11, G14) contained only some of the search terms and by investigating them for less than one minute each, we discarded them. The second example (G12) used all the queried APIs but consisted of only validation code; it did not use the retrieved data to generate any form of textual representation. The third example (G13) matched all the terms, but the first range highlighted by GCS contains references that the developer has actually commented out. The second range was promising, as it referenced all the query terms and converted various `ASTNode` elements into their textual representation. Unfortunately, this code fragment is also very long (1636 lines) and after spending 8 minutes investigating the method, we decided that there was too much extraneous functionality to effectively tease out the parts of interest. The fifth example (G15) also references all the query terms, matching across many methods (117). After 5 minutes we had located the best of these methods and concluded that it seemed to provide most of the functionality needed to complete the task; this method was chosen for integration.

Strathcona returned 10 examples. Examining the first five, we found that S11, S12, and S14 all contained partial matches that could be discarded in less than one minute each (Strathcona orders results in terms of the nature of the structural matches, not their quantity). The third example (S13) referenced all the required query terms and appended much of the information gleaned from the query, to a `StringBuffer` instance; in less than 5 minutes we were able to flag this example as worthy of integration. Example S15 was identical to example G15 from GCS.

Integration. Manually integrating G/S15 involved copying the method that seemed most relevant to our task into our code skeleton, resulting in 19 compilation errors. We modified the code by adding a new field (`buffer`) and removing an irrelevant method call; this reduced the number of errors to 5. We then noticed that we needed to copy another method (`printModifiers(List)`); this led to iteration that required four more methods to be copied into our skeleton, one after another. We also removed a call to an irrelevant private method. At this point the code compiled but we realized that without testing it we could not verify that it met our requirements. Execution of the code revealed its extensive use of visitors, which extracted a great deal more information than necessary for this scenario. Although the code might have been modified to eliminate the extraneous functionality, the quantity and potential impact of the changes required, combined with the uncertainty of what additional modifications would still be needed to complete the task, led us to abandon this example. Reaching this decision took 20 minutes.

To integrate G/S15 using Jigsaw we first copied the source code for that file from GCS into its own file in the workbench; initially this had many errors but by also importing the project in which it is found (the Eclipse `org.eclipse.jdt.core` package) we were able to resolve these errors; we had to perform this step as Jigsaw can only integrate fully compiling code. After selecting the source and target methods, Jigsaw copied the code (145 LOC in five methods, four plus the one selected) and modified it to fit its new context; after this, 27 compilation errors remained. After manually adding two fields that Jigsaw had failed to copy, 4 errors remained. We then deleted one local variable that had a name conflict with a field we added, resolving the final set of errors. Upon execution, the code displayed the same problems that we had detected during the manual treatment, so we again abandoned it. Reaching this decision took 8 minutes.

For the manual integration of S13, we copied first the key method highlighted by Strathcona (`buildMethod-Declaration(MethodDeclaration)`), resulting in 7 errors. We then copied `appendExtraDimensions(..)` and five other private methods, iteratively one at a time as we discovered they were needed. After these were copied, we were left with 4 errors which we realized were from code not relevant to the task. After testing the code we found that it provided the required functionality. Manual integration took 11 minutes for this example.

To integrate S13 using Jigsaw, we first copied over the file and imported the Eclipse `org.eclipse.jdt.core` package into our test project. After selecting the source and target methods, Jigsaw copied the code (214 LOC in seven methods, six plus the one selected for integration) resulting in 26 errors. After manually creating one `StringBuffer` field, 4 errors remained; these were resolved by removing 4 similar statements that were irrelevant to the task. After executing the code we found that it fulfilled the scenario. Jigsaw integration took 7 minutes for this example.

Scenario 2: Create an `ICompilationUnit`. A source file in Java is called a compilation unit, and the interface type `ICompilationUnit` represents these within Eclipse ASTs. The scenario involves creating an instance of an `ICompilationUnit` from a Java source file selected from the Eclipse package explorer; note that no simple constructor calls are available. The parents of the class under development are expected to be the class `ViewPart` and the interface type `ISelectionListener`. The task is expected to make use of the types `TextEditor`, `IAction`, and `IDocument`. The resulting code skeleton is shown in Figure 2. For the Strathcona query, we selected lines 11 to 14 of the code skeleton. The GCS query string "`ViewPart ISelectionListener TextEditor IAction IDocument lang:Java`" was created from the key details mentioned above.

```
 1  import org.eclipse.jdt.core.ICompilationUnit;
 2  import org.eclipse.jface.action.IAction;
 3  import org.eclipse.jface.text.IDocument;
 4  import org.eclipse.ui.ISelectionListener;
 5  import org.eclipse.ui.editors.text.TextEditor;
 6  import org.eclipse.ui.part.ViewPart;
 7
 8  public class XSnippetICompilationUnit extends
        ViewPart implements ISelectionListener {
 9    public void buildCU() {
10      TextEditor te;
11      IAction ia;
12      IDocument id;
13      ICompilationUnit icu;
14    }
15  }
```

Figure 2. Scenario 2 code skeleton.

*Location and selection.*GCS returned 31 examples. Of the first five, G21, G23, and G25 matched all the queried terms. Examples G22 and G24 only matched some of the search terms; a significant number of methods (23 and 13, respectively) had to be inspected to discover this fact, at high cost in terms of time (12 and 9 minutes, respectively), and these examples were subsequently discarded as irrelevant.

Strathcona returned 10 examples. Examining the first five, we found that all five examples contained only partial matches; using Strathcona's source code view, we were able to discard all five examples in less than one minute each. Strathcona's repository did not contain any relevant examples, including those returned by GCS.

Integration. The relevant examples G21, G23, and G25 were integrated into the code skeleton shown in Figure 2. The number of actions necessary to manually integrate the examples was 4, 13 and 11 respectively with 25, 33 and 15 minutes respectively required to do so. The Jigsaw tool required fewer actions from the developer (3, 7, and 4 respectively) with less time to perform the integration (3, 7 and 5 minutes respectively). In all three examples an action was needed to handled a static method call (`EditorUtility.getJavaInput(..)`) that required the developer to investigate a method declaration in another class; though the class name was shared by all three examples, each class was localized to its project.

The Jigsaw integration of examples G23 and G25 highlighted problems with the current implementation of the tool; the developer was required to perform actions to resolve dependencies on fields and methods from the methods depended upon by the original method (`setInput(..)`). For the indirect method dependencies, the Jigsaw tool was applied again selecting each indirectly depended-upon method declaration as the originating source method; indirect field dependencies required the developer to manually "copy & paste" them into the skeleton. For the Jigsaw integration, over half the time was spent setting up the examples to work with the tool for this scenario.

Scenario 3: Extract an `ITextSelection`. The Eclipse IDE captures modifications to source code editors in `TextEditorAction`s; in this scenario a developer must extract an `ITextSelection` given a `TextEditorAction`. This is essentially a call chain scenario. While this task is fairly straightforward, the `TextEditorAction` API contains 44 methods that can be directly called and hundreds that can be called indirectly, making an exhaustive search impractical. The code skeleton for this scenario is presented in Figure 3. For Strathcona, we queried lines 5 to 9 of the code skeleton, while the GCS query consisted of "`TextEditorAction ITextSelection lang:Java`".

```
1  import org.eclipse.jface.text.ITextSelection;
2  import org.eclipse.ui.texted.ITextEditor;
3  import org.eclipse.ui.texted.TextEditorAction;
4
5  public class TSAct extends TextEditorAction {
6          public void processSelection() {
7                  ITextSelection its;
8          }
9  }
```

Figure 3. Scenario 3 code skeleton.

GCS returned 322 examples; Strathcona returned 10 examples. Each location technique returned three examples that provided the developer's desired functionality (G32, G34, G35, S31, S32, S34). Implementations of `TextEditorAction`s tend to be smaller classes with limited functionality (ignoring inherited methods) which made assessing relevance an easy job for this task. Also, the functionality required itself was well contained, as it could be provided as a single call chain: `sel = (ITextSelection) editor.getSelectionProvider().getSelection();`

As the required code was so small, the needed fragments were easy to integrate manually. While using Jigsaw was still faster than the manual case in general at the actual integration, the set up time associated with these tasks overshadowed the time saved by using such an approach. While two of the examples returned by Strathcona used a more robust mechanism for returning the selection (by testing casts), the original paper [10] did not indicate that these were strictly necessary.

Scenario 4: Create a `TableModel`. In the Java Swing framework, the `TableModel` interface specifies the methods that integrate a tabular data model for the `JTable` graphical widget. This scenario involves the implementation of a `TableModel` to provide sorting and filtering functionality. From the description of the task, we created the code skeleton shown in Figure 4. For Strathcona, we selected the `SortFilterTable` constructor, lines 5 and 6 of the skeleton, and for GCS we queried with "`JTable TableModel Sort Filter lang:Java`".

```
1  import javax.swing.table.TableModel;
2  import javax.swing.JTable;
3
4  public class SortFilterTable implements
       TableModel {
5     public SortFilterTable(JTable jtable) {
6     }
7  }
```

Figure 4. Scenario 4 code skeleton.

Location and selection. GCS returned 2000 examples. All the first five examples matched all the queried terms. Further investigation of the examples revealed that only G41 had methods relevant to the scenario. Strathcona returned 10 examples. Examining the first five, we found that the first three examples contained only partial matches; using Strathcona's source code view, we were able to quickly discard the first three examples. The fourth and fifth examples (S44 and S45) contained methods relevant to the scenario.

Integration. We integrated the relevant examples G41, S44, and S45 into the code skeleton. During the manual integration of G41, it became apparent that the example was not relevant to the scenario task. The `SortFilterModel` class contained methods that followed a similar naming convention to the `TableModel` interface and the three methods found to be relevant contained dependencies on other methods that made explicit the `TableModel` use within the class as a type but not describing its creation. The integration was halted after 7 minutes and the application of the Jigsaw tool was not attempted (hence the "-" marks in Table 1).

We were able to successfully integrate S44 and S45 into the code skeleton. For S44 the developer was required to make 6 fewer actions manually compared to the use of the Jigsaw tool. Example S44 contained 8 methods and S45 contained 6 methods that were required to be integrated to complete this scenario; this required the developer to supply Jigsaw with each of the methods individually as input into the tool. Despite this overhead, the time spent manually integrating the source code for S44 and S45 was three times as much. However, there was a tremendous time cost of 61 and 58 minutes, respectively, associated with setting up the source (requiring significant functionality to be imported from the NetBeans IDE) to work with Jigsaw.

A.3. Observations and Analysis

While performing the case studies we investigated 40 examples and attempted to integrate the most promising 14 of these. From this experience we made a number of observations about the process of locating, selecting, and integrating these types of examples.

Classifying examples. The four scenarios we undertook were derived from the related literature, but in performing these tasks we observed that they fell into two categories. Scenario 2 involved what have been termed call chains, object instantiations [8], and method invocation sequences [10]; these involve short snippets of code that usually demonstrate how to access some functionality in an API. Conversely, feature-oriented scenarios require richer functionality; these are demonstrated in Scenarios 1, 3, and 4. In Scenario 2 we noticed that relevant call chain examples were easier to identify, occurred more frequently because their scope was broad (the same call chain can be used by examples of very different functionality), and were easy to integrate manually as the significant parts were generally short. Deciding that an example was relevant to a feature-oriented scenario was more difficult: as more significant functionality is required from them, they are generally larger and more specific. In general, we found that to really be able to tell if an example offered the required functionality for a feature-oriented scenario, the example had to be integrated into our skeleton and tested; this also helped us to ultimately determine which example was best for a task. For example, in G/S15 and G41 we believed that a given example was relevant, only to realize during integration that it was not. The various approaches that we discussed in Section 2 generally support call chain scenarios and are not effective in feature-oriented scenarios.

Location approach shortcomings. While we were selecting scenarios to use for these case studies, we often encountered situations where GCS would report a given result but an uninformative error message would be returned when we attempted to investigate the result. More significantly, Strathcona also demonstrated the shortcoming of its centralized repository approach, as in Scenario 2 the repository contained no relevant examples.

In addition, GCS sorts examples relative to the query terms that are sent to it, and performs lexical matching within entire files, selecting files where parts of identifiers and comments match the terms; Strathcona ranks examples according to the structural heuristics that are matched on the server. Neither location approach, or any we are aware of, can rank examples in the order that would be most conducive to the developer being able to reuse the example. For example, in G13 the example matched all the search terms but was also more than 1.5 kLOC in length with much extraneous functionality; this example would likely not have been as easy to integrate as another example that was < 100 LOC in length.

Manual integration shortcomings. We observed two main shortcomings of manual integration tasks. First, it was not easy to tell, through manual inspection, what dependencies the example code might have on the rest of its class or the system from which it was extracted. (This is consistent with an earlier study that found that developers often fail to identify source code dependencies using source code editors [6].) This meant that after we copied over some fragment of code, another fragment, method, or field would also be required, often iteratively. Second and more intrusive, the example's context was often different from our code skeleton: fields, local variables, and methods often had to be renamed. For small examples this was straightforward but for larger examples we had to be careful that we were not breaking how the code worked by modifying it.

Jigsaw integration shortcomings. Configuring an example so it could be integrated by Jigsaw often took more time than the actual integration; beyond our controlled environment, it is thus unlikely that Jigsaw would be used in end-to-end example reuse tasks. This setup burden arises because Jigsaw needs full bindings (to the declarations of types, methods, etc.) from the Eclipse AST in order to do its analyses well. For example, if an example file contained the call chain `one().two().three()`, without bindings the return type of `two()` could not be determined and thus the type upon which `three()` is invoked would be unknown. Jigsaw uses this information to ensure integrity when integrating fragments of source code, as two elements of identical type are more likely to correspond than two elements of differing type.

A.4. Summary

We found that locating, selecting, and integrating source code examples with existing tool support is approachable but generally cumbersome and slower than necessary. This burden led to situations where we feel that after spending time to integrate one example we would be unlikely to spend more trying to integrate alternatives—in a realistic development setting, at least.

We found 5 significant issues with using these existing approaches as end-to-end support for reusing examples: (1) example repositories that do not contain examples relevant to a scenario are not helpful to that scenario; (2) location techniques have to consider syntax and semantics to locate better examples for reuse; (3) located examples

GCS					Strathcona				
#	Example class	Methods		Time	#	Example class	Methods		Time
		P	TP				P	TP	
Scenario 1: Compute the signature of a `MethodDeclaration`									
G11	`EclipseCFG`	19	0	2	S11	`AccessAnalyzer`	1	0	1
G12	`Verifier`	22	0	5	S12	`DOMFinder`	1	0	1
G13	`ASTScriptVisitor`	21	0	8	S13	`SourceBasedSourceGenerator`	1	1	3
G14	`ASTConverterTESTAST3_2`	117	0	5	S14	`ImportRemover$1`	1	0	1
G15	`NaiveASTFlattener`	74	1	8	S15	`NaiveASTFlattener`	1	1	2
Scenario 2: Create an `ICompilationUnit`									
G21	`J2SView`	7	3	6	S21	`ImportResourcesAction`	1	0	<1
G22	`PHPEditor`	23	0	12	S22	`ExportResourcesAction`	1	0	<1
G23	`ASTView` (v2008)	14	4	8	S23	`AbstractMemoryViewPane`	1	0	<1
G24	`InstallOptionsDesignEditor`	13	0	9	S24	`Visualiser`	1	0	<1
G25	`ASTView` (v2004)	11	3	5	S25	`TaskList`	1	0	<1
Scenario 3: Access text selection									
G31	`IndentAction`	1	0	1	S31	`JoinLinesAction`	1	1	2
G32	`GoToNextPreviousMemberAction`	2	1	2	S32	`AddMarkerAction`	1	1	<1
G33	`PLEditor`	1	0	2	S33	`ClipboardOperationAction`	1	0	<1
G34	`RubyEditor`	2	1	2	S34	`BlockCommentAction`	1	1	3
G35	`GoToAction`	1	1	2	S35	`JavaMoveLinesAction`	1	0	1
Scenario 4: Create a `TableModel`									
G41	`SortFilterModel`	9	3	4	S41	`ReverseCallGraphPanel`	1	0	1
G42	`JTable`	37	0	8	S42	`TagHandlerPanelGUI`	1	0	2
G43	`TableRowSorter`	4	0	1	S43	`VisualArchiveIncludeSupport`	1	0	1
G44	`TransFilterTable`	7	0	2	S44	`TreeElAttribListCustomize`	1	1	2
G45	`SortedTableHelper`	14	0	5	S45	`FilterSetsPanel`	1	1	1

Table 1. Investigation of the results from GCS and Strathcona. The "P" column indicates the number of methods that the approach marked as relevant (positives); "TP" indicates the number of methods that we determined to be relevant to the scenario (true positives); "Time" indicates the approximate time we required to investigate the result, measured in minutes.

should be ordered relative to ease of integration with the developer's context; (4) overhead for moving examples into the developer's environment and resolving trivial issues has to be reduced or eliminated; and (5) the form of integration needs to consider factors such as: avoiding the alteration of APIs; and iterative issues from copying yet more code to eliminate dangling references.

B. Expanded Discussion

A naive response to the need for end-to-end support for example reuse tasks would be to take a couple of the existing approaches and combine them through a bit of engineering. In Brooks's phrasing [1], this could eliminate the *accidental* complexity of the end-to-end tasks but not the *essential* complexity. None of the existing tools were designed with these end-to-end tasks in mind. As such, their designs are not merely incomplete with respect to the end-to-end tasks, but they actively conflict in significant ways. This is not a criticism of the quality of the ideas or engineering that have gone into any of those tools (regardless of the fact that a few bugs were found); it is a recognition of a *mismatch* issue [5] that is not obvious when considering only a high-level view of each tool and its place in these end-to-end tasks.

On the other hand, the existing approaches are not completely off the mark: we were able to perform the end-to-end tasks, although this was more burdensome than it should have been. Of the available example location approaches, Strathcona is closest to what is needed for locating reusable, feature-providing examples; Google code search can be coerced for this, but it is a more general-purpose search engine that would require more work to fit well into end-to-end example reuse tasks (informally, we have noted that other code search engines suffer from similar issues). While approaches that specifically address call

#	Example class	Manual			Jigsaw			
		actions	size	time	actions	size	setup t	action t
	Scenario 1: Compute the signature of a `MethodDeclaration`							
G/S15	`NaiveASTFlattener`	8	177	20	4	167	5	3
S13	`SourceBasedSourceGenerator`	8	224	11	3	214	5	2
	Scenario 2: Create an `ICompilationUnit`							
G21	`J2SView`	4	56	25	3	56	5	3
G23	`ASTView` (v2008)	13	141	33	7	154	8	7
G25	`ASTView` (v2004)	11	80	15	4	87	6	5
	Scenario 3: Access text selection							
G32	`GoToNextPreviousMemberAction`	3	2	2	2	2	4	<1
G34	`RubyEditor`	3	2	2	2	2	3	<1
G35	`GoToAction`	3	2	2	2	2	2	<1
S31	`JoinLinesAction`	3	10	2	3	13	4	<1
S32	`AddMarkerAction`	1	1	1	2	1	4	<1
S34	`BlockCommentAction`	2	12	2	2	14	2	<1
	Scenario 4: Create a `TableModel`							
G41	`SortFilterModel`	3	-	7	-	-	-	-
S44	`TreeElementAttributeListCustomize`	8	209	21	14	215	61	7
S45	`FilterSetsPanel`	7	51	8	7	59	58	3

Table 2. Integration of the relevant examples into the code skeletons. The "#" column refers to the example number from Table 1; "actions" indicates the number of discrete actions that the developer had to perform (e.g., manually adding an import statement that resolved a dozen errors counts as one action); "size" indicates the LOC that were reused; "time" is the time taken to perform the integration, in minutes, and this is differentiated for Jigsaw into time to move the example into an actionable form ("setup t") and time to actually perform the integration ("action t").

chain scenarios are good at addressing call chain scenarios, these do not require developer interaction with concrete examples and are trivial to integrate—better support for integration seems unnecessary for this more narrow focus.

For integration, the assumptions made by Jigsaw are problematic; nevertheless, it outperformed the manual approach significantly when the setup time is ignored. Those assumptions are not foundational to Jigsaw, so using it as a basis for the integration portion of the end-to-end support seems reasonable at this point, as long as the means for significantly reducing or eliminating setup time is found. We know of no other approaches that could effectively integrate feature-providing examples.

How can the issues be overcome? A reasonable starting point for end-to-end tool support for example reuse tasks would appear to be combining Strathcona and Jigsaw. But as we pointed out above, engineering alone will not suffice to overcome the issues we identified in the case studies.

To eliminate the fact that Strathcona relies on a centralized repository that can easily become stale (which is why it could find no relevant examples for Scenario 2), two routes are possible. (1) A revised Strathcona could automatically retrieve examples from web-based code search engines and then analyze these according to Strathcona's approach to approximate structural context matching; however, care would need to be taken to maintain Strathcona's high speed performance. (2) A revised Strathcona could use the centralized repository as a cache, and only query the web-based search engines when a cache miss occurred; again, tuning such an approach to maintain performance would be important.

More work should be put into the selection phase to increase the chances of investigating the best integration candidate first. This could be achieved either through altering Strathcona to introduce one or more new heuristics [7, Section IV-C] and possibly to remove others, or by analyzing the examples returned by Strathcona and reordering them according to alternative heuristics. The latter option has the advantage of not altering Strathcona but the disadvantage that it cannot so easily add additional examples that Strathcona has chosen to ignore.

The process of integration should be better automated and more oriented towards the end-to-end task so that developers can more readily try multiple candidate examples to find the one that best meets their needs. Part of the difficulty arises from the fact that Jigsaw (and to a lesser extent, Strathcona) depend on the availability of resolved references (i.e., bindings). Bindings are necessary to allow Jigsaw to fully understand what is happening in the source code. For example, in the call chain `getActionBars().-getStatusLineManager().setMessage()` Jigsaw cannot interpret the return type of `getActionBars()` or `getStatusLineManager()` without having access to the AST bindings. To overcome this, a technique could be applied that allows partial program analysis to determine the likely resolution of a given reference. Dagenais and Hendren have recently introduced an algorithm for partial program analysis that does just that [3]; its existing tool support within the Eclipse IDE ought to make this a relatively painless exercise. Alternatively, the techniques could be made to depend on lexical identity when a given binding is unavailable. Which is the easier and more effective alternative remains to be seen.

Are the case studies biased and lacking in statistical significance? Our evaluation consisted of four author-performed case studies using two different location and integration approaches. While we did not try to explicitly account for learning effects (e.g., through randomization), we did try to report consistent times for tasks that were performed largely the same. While we cannot be sure that the case studies generalize, the purpose of our investigation was exploration: to determine whether it would be worthwhile to invest the considerable effort in explicitly supporting end-to-end example reuse tasks, and to determine what issues need to inform the design of that support.

Our investigation provides evidence that the end-to-end use of source code examples could be improved. The case studies were selected from scenarios published in the literature and they have pointed to shortcomings of our own tools. Thus, we have formulated a hypothesis (that overcoming the identified issues will significantly improve the end-to-end support) and outlined a plan for making it testable (overcoming the identified issues, as described above). Only at that point will it make sense to attempt a formal experiment and/or to obtain industrial feedback on the usefulness of the approach.

Do these case studies teach us anything beyond the domain of reusing examples? The reductionist tradition would suggest that we ought to identify a core set of small tools to be combined as needed to fulfill larger tasks. The accidental complexity inherent in the combinations could be overcome largely by designing and building them to certain standards—witness the prevalence of integrated development environments today. However, these case studies demonstrate how the complexity and richness of higher-level tasks can call for specialized end-to-end support; small and simple tools may be amenable to this need, but only if their behaviour can be constrained after-the-fact according to the larger-scale needs.

What is gained by providing end-to-end support? End-to-end support enables the developer to more quickly investigate potentially relevant examples, allowing them the possibility to use the time savings to investigate more examples and perhaps to discover better functionality or issues that were not otherwise apparent (like error cases). The fundamental difference between call chain scenarios and feature-provision scenarios is the richness involved: the more complex but non-standard the functionality, the more likely the need for in-depth investigation of multiple examples. Ignoring the details often leads to making mistakes in the details.

References

[1] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[2] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 214–225, 2008.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proc. ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, pp. 313–328, 2008.

[4] B. Dagenais and H. Ossher. Automatically locating framework extension examples. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 203–213, 2008.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, 1995.

[6] R. Holmes and R. J. Walker. Task-specific source code dependency investigation. In *Proc. IEEE Int'l Wkshp. Visualizing Softw. Underst. Analys.*, pp. 100–107, 2007.

[7] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[8] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pp. 413–430, 2006.

[9] J. Starke, C. Luce, and J. Sillito. Working with search results. In *Proc. ICSE Wkshp. Search-Driven Dev.: Users Infrastr. Tools Eval.*, 2009. To appear.

[10] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. IEEE/ACM Int'l Conf. Autom. Softw. Eng.*, pp. 204–213, 2007.