A Case Study in Simulated Concurrent Development and Evolution: Investigating the Theme Approach

Shafquat Mahmud
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
mahmud@cpsc.ucalgary.ca

Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
rwalker@cpsc.ucalgary.ca

Technical report 2004-765-30

1 October 2004

ABSTRACT

AOSD aims at improving key software engineering properties (such as traceability, comprehensibility, and evolvability) through the separation and modularization of crosscutting concerns. The majority of AOSD research focuses on individual software engineering activities (such as implementation or requirements) in isolation. One exception to this trend is the Theme approach of Clarke and colleagues, which considers the derivation of implementations from requirements through design. Evidence is currently meagre for or against the claims of this approach. This paper describes a case study involving the development and evolution of a benchmark system to evaluate these claims. Alternative decisions are examined to consider whether one or more feasible development processes exist in applying Theme. Lessons learned from the study are discussed for their generalizability to other scenarios.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

General Terms: Design, Experimentation, Languages.

Keywords: Aspect-oriented software development, Theme, Theme/UML, software evolution, concurrent development, case study, decision tree.

1. INTRODUCTION

Aspect-oriented software development (AOSD) promotes the separation and modularization of crosscutting concerns, in order to improve a number of key software engineering properties such as comprehensibility and evolvability [16]. Much AOSD research has concentrated on how crosscutting concerns can be separated and modularized—i.e., implementation issues; relatively little has considered how aspect-oriented systems should be designed. The Theme approach [5, 3, 6, 7, 8] goes beyond the question of how crosscutting concerns can be separated to consider when and why. It suggests that a separate design (called a theme) be developed for each requirement, and that these themes then be composed to form the complete system. The Theme approach claims to improve traceability, evolvability, comprehensibility, configurability, and concurrent development [3]. However, to date, the Theme approach has lacked evaluation of its core claims, concentrating instead on motivational examples to provide explanations of its workings.

Whether the Theme approach is best applied by explicitly separating crosscutting concerns immediately, or by implicitly dealing with them when they arise, is unclear. Both alternatives have been promoted [3, 1]. Perhaps neither would work well in practice; perhaps the alternatives are effectively equivalent; or perhaps each alternative has certain contexts in which it works best. This uncertainty reflects a debate within the AOSD community [13]: is an asymmetric model (where there is explicit base code and aspects are added to it) best, or is a symmetric model (where no implementation models are treated strictly as either base or aspects) best? Current evidence indicates that each model has its strengths and weaknesses [18], but discerning these deep issues beneath shallower issues of tool usability can be problematic. An imbalance in tool strengths and weaknesses can falsely skew perceptions. The apparent ability of the Theme approach to cope with either symmetric or asymmetric models [6, 7] might allow the two models to be evaluated on a fair and equal footing. But to reach that point, we need evaluation of the strengths and weaknesses of the Theme approach itself.

As a preliminary step in such evaluation, this paper describes a case study in applying the Theme approach in developing and evolving a benchmark system. Since Theme is essentially a descriptive development methodology and not prescriptive, we have considered the many possible paths through the tree of development decisions to determine how (or whether) Theme can be applied in practice.

The paper is organized as follows. Section 2 describes background and related work on AOSD and the Theme approach. Section 3 summarizes the case study and the lessons learned from it. Section 4 discusses the strengths and weaknesses of our study itself and how future evaluations can proceed.

The contribution of this paper is to provide a preliminary study into the claims of the Theme approach regarding traceability, evolvability, and concurrent development, and to consider the effects of alternative decisions in applying Theme when it is not prescriptive.

2. BACKGROUND AND RELATED WORK

The asymmetric and symmetric models to AOSD are exemplified by the implementation languages AspectJ [15] and Hyper/J [21]; both build atop Java.

AspectJ permits crosscutting concerns to be encapsulated in class-like constructs called aspects, which are explicitly sepa-

rated from a standard object-oriented class hierarchy, called the base code. Aspects are most often described as separating non-functional requirements, such as synchronization or distribution, but they can also separate functional requirements, such as the implementations of individual features or collaborations. Each aspect defines the points in the base code at which behaviour should be inserted. These join points can be either static points in the source code, or descriptions of more dynamic properties that occur at run time; particular method executions, field accesses, and object creations are typical examples. The behaviour to be inserted at specific join points is described in method-like constructs called advice. The AspectJ compiler combines aspects and base code to construct functioning bytecode.

Hyper/J is a language that extends an older AOSD implementation approach called subject-oriented programming (SOP) [12]. In SOP, a system is considered to be a composition of separate implementation models, called subjects, each of which represents the system from a particular perspective. Portions of a given object-oriented class would be represented in each subject according to the perspective there. For example, consider a Person class that is to represent a student who is also an employee; students and employees are different perspectives on this class that can be represented separately in two subjects. A subject-oriented compiler must combine these perspectives to form a functioning system. Hyper/J extended this idea to recognize that subjects and classes are two, orthogonal dimensions that describe systems and that additional dimensions can be of interest in some tasks; this is the notion of multidimensional separation of concerns [22].

The terminology within the AOSD field has evolved over time. The ideas underlying AspectJ were and still are often treated synonymously with the term aspect-oriented programming (AOP), and not as one particular flavour of AOP. Historically, this is understandable since AOP was described as distinct from SOP [16]. The encompassing term advanced separation of concerns (ASOC) was used to describe these and similar techniques, but was later abandoned in favour of AOSD.

Some work has evaluated the claims of AOSD implementation approaches. Murphy and colleagues [18] provide some initial evaluation of the abilities of AspectJ and Hyper/J to refactor crosscutting concerns in existing systems. Various case studies have been conducted in the aspect-oriented implementation of systems (e.g., Kersten and Murphy [14]). Hannemann and Kiczales [11] demonstrate how the implementations of certain design patterns can be separated via AspectJ. Walker and colleagues [23] performed a series of semi-controlled experiments that considered whether the effects of explicit separation of crosscutting concerns were beneficial or harmful when implementing, understanding, and evolving systems; the results were mixed, depending on details of the situation. Baniassad and colleagues [2] performed similar studies later in an industrial context, and found that non-separated crosscutting concerns represent an impediment to software evolution tasks in practice. Kienzle and Guerraoui [17] argued that separation of crosscutting concerns can be harmful under some situations, reinforcing earlier findings. Coady and Kiczales [9] investigated AspectJ-style separation of some crosscutting concerns in an operating system and the effects of this on the re-visited evolution history of that system; the results indicate benefit from the separation.

The ability to separate crosscutting concerns within an implementation does not aid in identifying crosscutting concerns or in deciding how to design an aspect-oriented system. Some work has been undertaken to identify crosscutting concerns at the requirements level and to provide modelling languages with which to represent aspect-oriented designs. Unfortunately, relatively little work has considered how to derive those designs from requirements in the first place—i.e., full lifecycle issues [7]. Theme [5, 3, 6, 7, 8] is a notable exception.

Theme derived from the concepts of SOP, and was originally called subject-oriented design [5]. Theme suggests that each requirement should be designed separately in a model called a theme (analogous to SOP subjects). Explicit relationships are then defined to specify how the design elements in each theme correspond and how they should be composed. Theme Design provides an extension to the Unified Modeling Language (UML), called Theme/UML, to permit modelling of themes. We examine details of Theme/UML in Section 2.1. Recently, Baniassad and Clarke [1] have provided Theme/Doc, an approach for identifying crosscutting concerns in structured, natural language, requirements specifications and using these to drive the design modelling in Theme/UML. This application of Theme favours the asymmetric model of AOSD, resulting in crosscutting themes and base themes. Explicit identification of crosscutting concerns from the requirements specification certainly has some potential benefits, but it also has potential drawbacks. For example, evolution of the requirements could result in widespread changes to the identified crosscutting concerns, negating the value of modularizing the original crosscutting concerns: there is no guarantee that every crosscutting concern will have been identified in the original system, or that changes to the set of crosscutting concerns will be readily pluggable or unpluggable in an asymmetric approach. But as Clarke originally defined Theme [3], such explicit identification should not have been necessary. One should have been able to perform a feature-oriented decomposition, and any crosscutting concerns could be dealt with at composition time. Lacking published evidence to the contrary, this possibility remains viable.

The Theme approach claims [3]: (1) if a requirement changes, its design can be changed in isolation from the designs for other requirements; and (2) if additional requirements are needed, they can be designed and composed without altering the original themes. These claims are effectively those of the benefits of locality and pluggability. Themes can be composed at design-time and implemented in an object-oriented fashion, or the themes and the composition relationships between them can be implemented directly via an AOSD implementation language. Following on these arguments, traceability from requirements to implementation ensues, evolvability is improved, and concurrent development of themes becomes possible, since any inconsistencies between themes can be eliminated at composition time.

Parnas [19] defined the goals of modularization as improved comprehensibility, evolvability, and ability to perform concurrent development on a system. Theme's claims align well with these goals. Communication overhead is a critical issue in a concurrent team development environment. Avoidance of communication overhead among development teams can improve distributed team development while reducing its cost. After-the-fact composition of conflicting design models would presumably permit communication overhead to be avoided by Theme. The claims of Theme would be valuable targets if they were achievable.

2.1 Theme/UML

Theme/UML [4] extends standard UML to support separated, composable design models (themes). Each theme is intended to "capture one requirement" as a separated, object-oriented design model. Themes are modelled as stereotyped packages.

Different perspectives on a system, and hence the themes used to model those perspectives, will typically overlap. To form a complete model of a system, the differing perspectives of individual

themes must be reconciled through the specification of composition relationships. A composition relationship will indicate which model entities in two themes correspond to each other and how they should be composed. The two most common relationships are merge and override. Merge indicates that the composed entity should be a true combination of the original entities. Override indicates that one should be discarded in favour of the other. Integration rules may be defined on composition relationships to avoid the specification of individual composition relationships for every entity present. A simple rule is "match-by-name" which causes the merge of all entities of identical kind with identical names. Explicitly crosscutting themes can be modelled by parameterizing an ordinary theme via an approach based on UML templates [6, 8]; as we will not be using this feature in this paper, we will not discuss it further.

Figure 1 shows an illustrative example of Theme/UML, where two themes (`USER` and `PORT`) are to be composed by according to the composition rule merge-by-name. The resulting, composed theme is shown in Figure 2. Each theme is designed as a stereotyped UML package, containing various classes and their interrelationships (in addition, interaction models and various other UML elements can be contained in themes). In the before-composition model, the dotted arrow connecting the `USER` and `PORT` themes represents a composition relationship, indicating that the two themes are to be composed; the presence of arrowheads at both ends of this arrow indicates that the themes are to be merged, rather than one replacing the other. The annotation `match[name]` on this arrow indicates that all elements within the package with identical names (judged in a hierarchical fashion) are to have a single, corresponding element in the resulting, composed theme. In this example, the names of three classes are identical between the two themes: `ControlConn`, `Executer`, and `ArgSynErrorReply`.

If the model contained no further composition relationships, the resulting, composed theme would contain seven classes: `ControlConn`, `User`, `Executer`, `UserLoggedInReply`, `ArgSynErrorReply`, `Port`, and `OkReply`. However, the correspondence of the names of the two `Executer` classes is coincidental. Not wanting these two to be merged into a single class, a lower-level composition relationship is specified between them indicating `dontMatch`. This directive prevents these classes from being merged.

We also see that the two `ArgSynErrorReply` classes are identical. This is not coincidental. Were we to simply permit these twins to be merged, any invocation of the methods `getCode()` or `getMessage()` would invoke both of the original implementations, in some unspecified sequence. Since this double invocation is not necessary or desirable, we can simply have one copy of the class replace the other outright. This override composition relationship is indicated with a one-way dotted arrow. Any elements contained within the classes that are also identically named will likewise have one copy replace the other, because of the presence of the composition rule `match[name]`.

In the composed theme (Figure 2), we see eight classes resulting from the specified composition: each `Executer` class has been maintained in its original form, but renamed to avoid name collisions. Relationships between classes also correspond to the originals; for example, in the `PORT` theme, the `Port` and `Executer` were associated, so their corresponding classes in the composed theme (`Port` and `PORT_Executer`) will also be associated.

This is a relatively uncomplicated example of the use of Theme/UML. Unfortunately, themes are not always so readily composable, as we shall note in the next section.

3. CASE STUDY

This section describes a case study in applying the Theme approach in developing and evolving a benchmark system [10].

This case study pursued two goals: (1) to evaluate the Theme approach claims of improving traceability, evolvability, comprehensibility, configurability and concurrent development without communication overhead among different teams; and, (2) to consider how (or whether) Theme can be applied in practice, since many paths through the tree of development decisions are possible. Any feasible path through this decision tree may exemplify a process that is generally applicable, and any infeasible path may exemplify a general pitfall.

The case study consisted of the development and evolution of a File Transfer Protocol (FTP) server via the Theme approach. FTP is described in an informally structured, natural language, requirements specification document (RFC 959) [20]. We selected FTP as the benchmark system [10] to investigate because it is well-understood and small enough to analyze, but large enough to display difficulties involving traceability, evolvability, comprehensibility, and configurability.

FTP is a stateful protocol, involving the establishment of a control connection between a client and server. Files are communicated over a possibly transient, separate data connection. The client issues string-based commands to the server, which responds with reply codes (indicating success, failure, enter password, etc.); some commands initiate file transfer. Each command consists of a four-letter mnemonic followed by arguments whose syntax depends on the command being issued. FTP defines state machines for the legal sequences of some commands. See RFC 959 for further details [20].

We began our study by developing the "Minimum Implementation" of an FTP server as required by RFC 959. Later versions of the system added features to this base. Version 2 added the FTP authentication protocol, Version 3 added passive mode features, Version 4 added the user login functionality with different accounts, and Version 5 added help information options to the system. Each version was designed and implemented without consideration of any remaining features described by RFC 959, to simulate the need to accommodate unpredicted changes.

Throughout the life cycle of the system, we explored alternative development decisions that different development teams might make. Figure 3 outlines the tree of these development decisions, which we traversed through the development and evolution of the system. The remainder of this section is structured as follows. Section 3.1 provides an overview of the decision tree shown in Figure 3. Section 3.2 describes the details of the decision tree examining consequences of alternative decisions at each level. Section 3.4 overviews the resulting evolution steps in constructing Versions 2 through 5 of the system, each of which involved additional traversals of the decision tree from root to leaves. Section 3.3 describes issues in composing the implementations of each theme. Section 3.5 considers the feasible and infeasible paths through the tree and discusses the lessons learned.

3.1 Overview of Decision Tree

The development of the system began from the "Minimum Implementation" section of RFC 959, and thus, this document forms the root node of the decision tree. The first decision needed was whether a structured specification was required to develop a system with the Theme approach—and what form it should take—or whether the informal specification was sufficient to proceed with the development process. The second decision involved determining ways to decompose a system specification (structured or

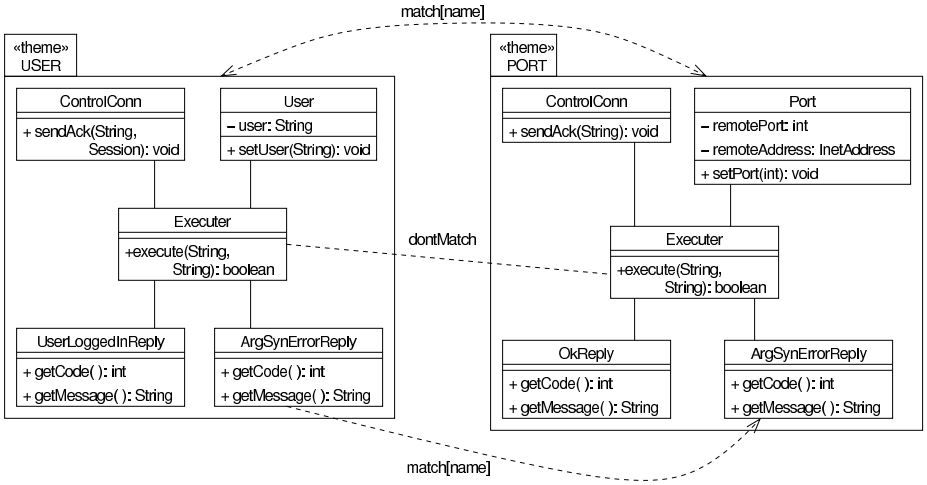

Figure 1: An illustrative example of the composition of themes in Theme/UML. Shown are two themes that are to be composed.

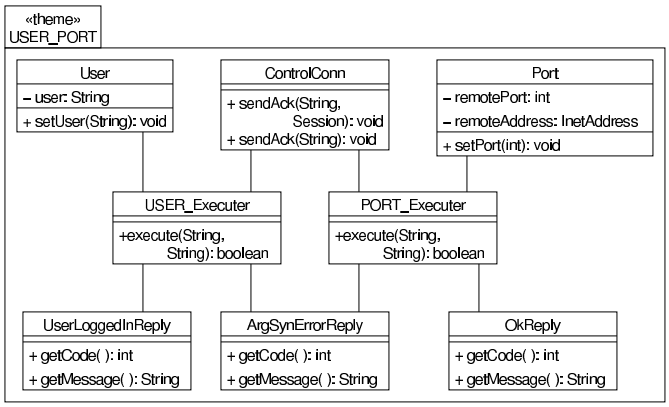

Figure 2: The resulting, composed theme.

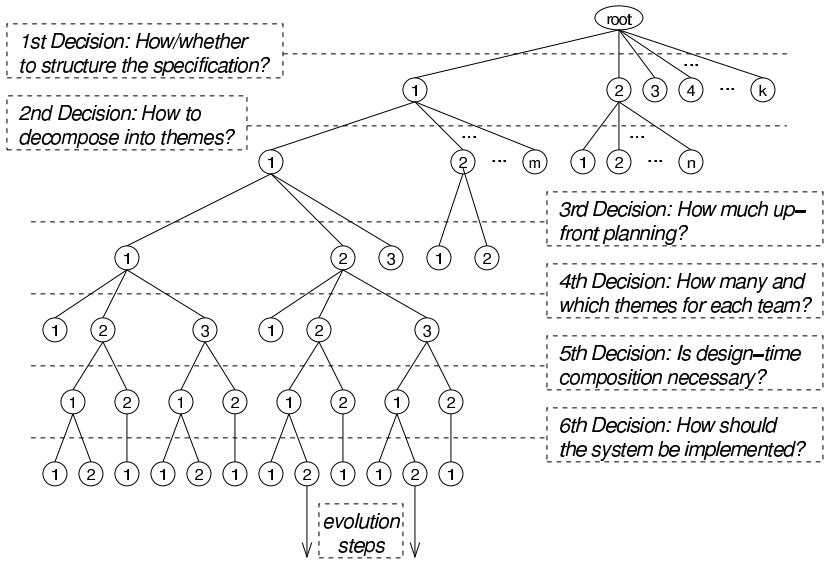

Figure 3: The tree of decisions involved in applying the Theme approach. Each node is labelled with its path-local number; the global label for a node is found by traversing the tree from the root to that node. For example, the bottom-leftmost leaf has path-local label 1, but a global label 1.1.1.2.1.1.

unstructured) into themes. The third decision determined if any up-front planning is required to support concurrent team development with the approach. If some planning up-front were required, then the decision would involve determining how much planning is needed to ensure different development teams can work separately without communication overhead to develop a system. The fourth decision was also related to this: consideration of how themes should be distributed among teams, since different task assignments of themes among development teams might affect the resulting development process significantly. We considered our fifth decision to be whether a design-time composition of the themes by different teams is required. Design-time composition can be performed to ensure that the implementations by all teams meet some degree of uniformity for later composition of implementations. Alternatively, the design and implementation of themes (with or without up-front planning) by each individual team might suffice for the composability of those models into a fully-functioning system. The sixth and final decision was to select an implementation strategy that would satisfy the constraints imposed by the particular path through the decision tree that brought us to this decision.

3.2 Details of Decision Tree

The root of the tree starts from the informal specification of the system. As RFC 959 contains the informal specification of a complete FTP server, for Version 1 of the system we extracted an informal specification from RFC 959 containing only those details needed for this version. This extraction involved copying all the details from RFC 959 that correspond to the "Minimum Implementation" of FTP to produce an informal, natural language specification.

The remainder of this section discusses the consequences of alternative decisions at different levels of the tree. This discussion is formatted as a breadth-first traversal; a depth-first traversal will be used in Section 3.5.

3.2.1 1st Decision: Structured Specification?

The first decision, taken at the root node, was to consider if a structured specification is required to derive themes, or if themes can be derived decomposing a natural language description of the system. Many possible paths can be considered here. One can continue with the informal specification (Node 1). One can create a traditionally structured, natural language specification (Node 2). Different requirements engineering techniques like formal modelling, use case modelling, repertory grids, concept maps, etc. (Nodes 3 through k) can be ways to derive a structured specification.

Formal modelling can be a useful approach to mathematically validate a system, but in most industrial contexts, it is considered an impracticable technique because of its high cost. Therefore, we have not attempted to proceed to Node 3.

We only consider Nodes 1 and 2 further within this case study. Issues involving other requirements engineering techniques (Nodes 4 through k) are discussed in Section 4.

3.2.2 2nd Decision: Decomposition into Themes?

The second decision was how to decompose the system specification (structured or unstructured) into themes. We explored paths through Nodes 1 and 2.

Several paths can be considered from Node 1, the informal, unstructured specification. We first attempted a feature-oriented approach to derive themes (Node 1.1). We identified features of the system and extracted the description of each feature from the specification. Each extracted description described a part of the system and can be considered as system feature/requirement/task that is to be mapped to a theme. We extracted 11 features in this way, 9 of which involved an individual FTP command (USER, PORT, TYPE, MODE, STRU, RETR, STOR, NOOP and QUIT) and the other 2 were for establishing connection with a user and for interpreting his/her requests. Different possible mappings between the features/requirements/tasks and themes are possible; as suggested by Clarke [3], a one-to-one mapping can ensure traceability compared to a one-to-many or a many-to-many mapping. One-to-many and many-to-many mappings both tend to cause scattering and tangling of the sub-parts (requirements/features/tasks) across different themes. Therefore we followed a one-to-one mapping and mapped the 11 identified features/requirements/tasks into 11 themes (Node 1.1).

An alternative path can be considered from Node 1, on which we can attempt to extract themes directly from the informal specification, not bothering to produce separate descriptions for each of the themes (Node 1.2). That would leave the development teams with only the name of each feature (the same 11 names identified for Node 1.1) and the complete, original specification from which to work. This could provide each team with the flexibility to interpret how much of the original specification that a feature involves and that the corresponding theme is supposed to encapsulate.

Presumably, alternative means of decomposition might be discovered that we have not considered (Nodes 1.3 through 1.m). We justify why our means is likely to be representative of a typical approach, in Section 4.

To decompose themes from a structured, natural language specification (Node 2), multiple alternatives can also be considered. An example is to consider the immediate identification and separation of crosscutting concerns into crosscutting and base themes (Node 2.1) as proposed by Baniassad and Clarke [1]. In this approach, a theme may consist of multiple system requirements that are common to a particular concern, not the one-to-one mapping between the requirements and the themes that would ensure traceability, as originally claimed by Clarke [3]. This approach also requires up-front work to identify different possible concerns to derive themes before different teams can be assigned particular themes to work with. Moreover, for every version of the system, the theme decomposition may require significant restructuring. Non-crosscutting requirements may become crosscutting in a new version; additional crosscutting requirements may not be plug-compatible with the current theme decomposition. The need for such restructuring would likely negate the benefits of a theme-oriented decomposition. However, we have not attempted to evaluate this path further to determine whether negative consequences would occur in practice. Such a study remains future work to evaluate how much it can fare with respect to key software properties like traceability, comprehensibility, and evolvability in a concurrent team development environment.

Other ways of deriving themes can be considered from Node 2. A path can be considered that maps every requirement in the structured specification into a theme, without consideration of explicit separation of crosscutting concerns (Node 2.2). However, this approach is similar to the path from Node 1 to Node 1.1, which we discuss further in Section 4. That additional alternative paths exist from Node 2 could be claimed (Nodes 2.3 through 2.m). Presumably, some of these might vary the granularity of decomposition, i.e., sub-requirements might each become a separate theme. Such fine-grained decomposition seems pointless in this benchmark system, as the resulting themes were already quite simple. Such approaches might become more viable as the system under consideration scales up.

3.2.3 3rd Decision: Up-Front Planning?

The third decision determined if any up-front planning is required to support concurrent team development with the approach. If some planning up-front were required, this decision would also involve determining how much planning is needed to ensure that different development teams can work separately while minimizing communication overhead in developing a system concurrently. We consider three alternatives from Node 1.1 and two from Node 1.2.

First, the themes derived in Node 1.1 can each be assigned to a different team without any up-front planning, such as interface definitions or naming conventions (Node 1.1.1). Each team is to develop its theme according to its specific descriptions, however they see fit. This approach can, of course, result in numerous naming/signature conflicts among the themes designed by different teams, as there can be no expectation that different teams will name identical concepts identically (e.g., `ControlConn` versus `ControlConnection`) or different concepts differently (e.g., the two `Executer` classes described in Section 2.1). Naming conflicts can be resolved during composition through fine-grained but straightforward composition relationships, as Theme claims. Disagreements in signatures are less trivial to cope with. Consider the following example of how we explored this path in the case study. We derived designs for `USER`, `TYPE` and `RETR` command themes to simulate the effects of differing (and somewhat incompatible) design decisions being made by independent teams working on each theme. The `USER` theme expects to be invoked only in the case of a valid `USER` command, whereby the argument passed to the theme's entry point should contain the arguments of the received `USER` command. On the other hand, the `TYPE` theme expects two arguments, the command (i.e., `type`, case insensitively) and its original argument string. This theme checks the command first, processing the argument only if the command corresponds to "`type`". And finally, the `RETR` theme expects to receive the raw client request and to parse this to retrieve the command and its arguments.

The second alternative from Node 1.1 is to perform some planning of themes up-front before each team begins concurrent development. In this way, fewer conflicts should occur at composition time, thereby reducing composition-time overhead. The question then becomes whether the cost of up-front planning is more than made up for by the savings accrued through avoiding composition-time conflict resolution. The other issue of concern here is to determine how much planning up-front is sufficient and feasible. Two paths can be considered with the up-front planning approach: one with lightweight up-front planning (Node 1.1.2) and the other with heavyweight up-front planning (Node 1.1.3).

The goal of lightweight up-front planning is to provide just enough information in the descriptions of different themes so that the teams developing them can understand what constraints exist on the boundaries of their respective themes—e.g., what the themes can expect from the system for their invocations, or what they are expected to return to the system. The conflicts described above between the `USER`, `TYPE` and `RETR` themes can be resolved up-front with the lightweight up-front planning approach. In attempting to conduct lightweight up-front planning, we added extra information to each command theme's description so that the themes would expect a common data format at the expected entry point from the system for each. The system architect, who decomposed the system specification into features (later mapped into separate themes), should already know enough detail about the system to determine this extra information with little additional effort.

The other path from Node 1.1 considers the consequences of performing heavyweight up-front planning (Node 1.1.3). While the lightweight approach deals with adding just enough extra information to the description of each theme to provide some uniformity to their entry points, the heavyweight approach aims to avoid all possible conflicts a priori, so that composition of themes would be straightforward. Simple ideas like defining naming conventions or common data formats would not be too onerous to realize, as with the lightweight approach. However, these would not suffice to eliminate significant conflicts and in exploring other paths, we found that name conflicts are relatively trivial to resolve during composition; the cost of spending too much time up-front should far out-weigh the cost of conflict resolutions during composition. Additional ideas like sketching out the themes prior to detailed design might be pursued, but such sketches would not converge to a conflict-free composed model any more than traditional stepwise refinement approaches [24] avoid the pitfalls of the Waterfall model of development: non-convergent iteration and slipped schedules. The attempt to resolve all possible conflicts up-front seems to require too much work to be considered a feasible approach. Considering this an infeasible path, we have not attempted to proceed to Node 1.1.3.

From Node 1.2, where each team is left with the original informal specification and the name(s) of the theme(s) to which it is assigned, two paths can be considered based on the use of up-front planning. First, we can choose not to perform up-front planning (Node 1.2.1). In our case study, we could interpret themes in various ways from the informal specification all of which seemed reasonable based on the names of the themes; where the themes ended was unclear, as they lacked objective boundaries. Unfortunately, without more information about a theme than just its name (or similar, short description), there remains the possibility that themes implemented by different teams will overlap in significant details or leave significant details unimplemented. Overlap indicates inefficiency in the concurrent development process as effort is duplicated amongst teams. Gaps indicate that requirements have remained unmet. At composition time, these overlaps and gaps might be detected. In fact, at a different node in the decision tree (see Section 3.2.5), we did detect unspecified requirements at composition time, which were then patched over. We have found that composition of themes developed in this approach can be expensive in consideration of the number of conflicts to be resolved and the number of missing features to be added during composition; thus, a path through Node 1.2.1 can be considered an infeasible one.

An alternative path from Node 1.2 can consider the use of some up-front planning (Node 1.2.2) to combat the problems of theme development, as just discussed. We have tried to determine what information can be passed to teams about different themes so that the problems of overlaps and missing details can be reduced to a reasonable level. But as we have discussed, the main problem with developing an individual theme from a system specification—in the absence of a specific description of the feature it encapsulates—is how to determine its boundary. Thus, up-front planning would necessarily involve some amount of determination of those boundaries, which would be effectively identical to Node 1.1. Thus, we do not proceed to or beyond Node 1.2.2.

3.2.4 4th Decision: Assignment of Themes?

The fourth decision issue was to determine how many and which themes to assign to each team. The remaining paths that are feasible and that we wish to investigate pass through Nodes 1.1.1 and 1.1.2. Three alternative paths can be considered from these nodes, based on an identical decision.

It can be assumed that there would be fewer numbers of conflicts among the themes developed by a single team; the most de-

sirable path would be to assign all the themes to a single team (Nodes 1.1.1.1 and 1.1.2.1). But this is an unrealistic solution as the advantages of concurrent development are immediately negated. These paths are not considered further.

Some random distribution of themes among teams could be performed based on an estimate of the work involved in developing a given theme and each team's available person-hours (Nodes 1.1.1.2 and 1.1.2.2). If conservation of effort is important, distribution of themes based on their similarity (Nodes 1.1.1.3 and 1.1.2.3) may make more sense. Similar themes being developed by a single team could be designed similarly, resulting in fewer numbers of conflicts at composition time, and thus easing the composition effort. The issue here is whether "similar" themes can be easily identified for separation, and whether errors in this identification (either falsely identifying a similarity or failing to identify an actual similarity) will cause harm.

In our case study, separation of themes that were similar was not difficult from their descriptions; this involved only 11 themes to consider, of course. We assigned `RETR` and `STOR` command themes to one team as they seem similar in functionality, `USER` and `TYPE` themes to one team and so on. Through the other path with random distribution of themes, we have tried to consider different designs of themes by different teams in our simulated concurrent team development environment.[1] In following these four paths involving random distribution versus similarity-based distribution, the only apparent difference was that random distributions tended to increase name conflicts; however, these conflicts were straightforward to resolve at composition time. Thus, all four remain feasible at this stage.

3.2.5 5th Decision: Design-Time Composition?

The fifth decision was to consider whether a composition of design themes is required prior to implementation. A design time composition can help verify that all conflicts can be resolved or that deficiencies in functionality exist. However, a careful decomposition of a system into themes may not require a design time composition to verify a system. Thus from each node x where $x \in \{1.1.1.2, 1.1.1.3, 1.1.2.2, 1.1.2.3\}$, two paths can be considered: one applying design-time composition of themes (Node $x.1$), and the other doing without it (Node $x.2$).

In our case study, design-time composition (all Nodes $x.1$) identified that a feature was missing. The themes that had been identified for design were derived from a feature-oriented decomposition of the informal, natural language specification (Node 1). That specification did not make explicit that the system needs to support multiple users in stateful sessions, although this requirement resides implicitly in the document. When we attempted design-time composition, this missing requirement manifested itself as a lack of any theme that would have been appropriate to compose directly with the `EstablishConnection` theme. This prompted us to investigate, at which point we recognized that between the establishment of a connection and interpretation of individual FTP commands lay a significant gap with no way to determine from whence the commands came. We then added a 12th theme to the system (`Session`) and assigned it to be developed by a separate team. The function of the theme was to provide a separate session for each user who gets connected to the system and all requests from that particular user would be processed within the session. This new theme was added to be developed in parallel with the others, but after up-front planning had been conducted. Nevertheless, the presence of no more than lightweight up-front planning did not require iteration back to an earlier node in the tree because of this discovery.

We continued the paths through the Nodes $x.2$ to confirm whether or not their lack of design-time composition would cause later difficulties.

3.2.6 6th Decision: Implementation Strategy?

The sixth and final decision was to consider which approach to follow to implement the design themes into a fully functioning system. Two approaches were possible: implementation of the composed design model (Nodes $y.1$ for $y \in \{1.1.1.2.1, 1.1.1.3.1, 1.1.2.2.1, 1.1.2.3.1\}$) or implementation of each theme independently followed by an implementation-time composition (Nodes $y.2$ for $y \in \{1.1.1.2.1, 1.1.1.3.1, 1.1.2.2.1, 1.1.2.3.1\}$ and Nodes $z.1$ for $z \in \{1.1.1.2.2, 1.1.1.3.2, 1.1.2.2.2, 1.1.2.3.2\}$).

Direct implementation of the composed design model would be a waste of the whole effort with the Theme approach. The result would just be an object-oriented implementation of the system as a whole. The claim of traceability between the requirements and the implementations would not be met. Changes to any single theme would result in modifications to scattered and tangled code. Although the severity of this problem might vary from system to system, it clearly would not represent the best application of the Theme approach. Thus, we do not consider the Nodes $y.1$ further.

Paths to the remaining nodes consider implementation of individual themes by different development teams and composing these implementations later by the architecture team. The Nodes $z.1$ demonstrated at this point the problem of not having verified the completeness and composability of the themes via a design-time composition. The missing requirement manifested itself, and had to be added after the fact. Since this additional theme would then have needed to be developed in sequence with the other themes, and not in parallel, the benefits of concurrent development with Theme were decreased. This approach would also have delayed noticing any serious design conflicts until after implementation was complete, also introducing the need for iteration prior to delivery of a single system version. We did not encounter this latter difficulty in our case study, but presume it to remain a possibility.

Of the remaining Nodes $y.2$, Nodes 1.1.1.2.1.2 and 1.1.1.3.1.2 were mildly problematic due to the lack of any up-front planning. Inconsistencies in naming conventions and differing expectations of preconditions on the entry points to independently developed themes had to be reconciled at composition time. The feasibility of these paths through multiple evolution steps might remain stable as only mildly problematic even as systems scale up. Thus, they represent less-attractive alternatives that can be considered for future study.

This left two nodes (Nodes 1.1.2.2.1.2 and 1.1.2.3.2) as the representatives of the paths that seemed most feasible up to this point. For implementation of the themes by different teams, we chose Java as a standard language for object-oriented implementation. For composition of the themes (implemented as java packages), various AOSD implementation languages/tools such as AspectJ or CME[2] might be used. For our study we chose to apply AspectJ, as the most commonly used and so far considered a standard AOSD tool. In addition, we chose to use AspectJ (which supports an asymmetric AOSD model) in composing theme implementations designed with a symmetric model in mind as an interesting test of the cross-compatibility of the two models. Similarly, paths through Node 2.1 (the approach of Baniassad and Clarke [1]) that design themes with

[1]The description and design of the themes can be found at http://www.cpsc.ucalgary.ca/~mahmud/theme.htm

[2]The Concern Manipulation Environment (CME) has superseded Hyper/J.

the asymmetric AOSD model in mind might apply a symmetric implementation tool to consider the opposite cross-compatibility; this remains future work.

The two paths in this level differ only in the distribution of the themes in the 4th decision; one considers random distribution of themes among teams and the other considers distribution of themes based on their similarities. In our case, we have found little difference between the approaches; both have considered descriptions of themes with lightweight up-front planning as discussed earlier. Since both of these paths provide themes that have been described with specific boundaries, it should not be surprising that the designs of the resulting themes do not differ much. We have found that differences in both the paths are minor, mainly involving shared class and method names. If done by the same team, the shared names can be uniform. We have also found that these naming conflicts do not matter much as they can be resolved in a straightforward manner during composition. In the end, the implementation of the first version of the system in both the paths proved to be much the same.

3.3 Composition Strategy for Implementation Models

For composing themes implemented as Java packages, we used AspectJ as the composition tool. Design-time composition in the Theme approach allows for merge and override integrations and composition relationships like match-by-name [3]. These integration mechanisms require the combination of behaviours from multiple classes in different themes, into individual, composed classes belonging to the composed theme. AspectJ does not provide any direct support for these mechanisms, since it is based on the asymmetric AOSD model for adding crosscutting behaviour to base code. We devised a technique to emulate the composition mechanisms of the Theme approach with AspectJ; the process is discussed below.

We represent themes as Java packages. For each composition relationship between themes, we create a new package to contain the resulting, composed theme. The classes from each theme participating in that composition relationship are copied into this new theme. Naming conflicts between classes from different themes are resolved by altering the names of the copies of these classes, prepending the name of the theme and an underscore to the original name of the class. If any of these classes are to be integrated according to the composition relationships defined, an additional, empty class is added to the new theme. An aspect will then be defined to introduce methods into this new class, and to advise these introductions to invoke each of the original methods.

Figure 4 shows a composition specification of two FTP themes from our case study; Figure 5 shows the resulting, composed theme. The themes participate in a composition relationship of match-by-name. Two classes (`Server` and `ControlConn`) of each theme are affected by this composition relationship. According to our technique, in our implementation we renamed them (as `EstablishConnection_Server` and `Session_Server`, and `EstablishConnection_ControlConn` and `Session_ControlConn`) before we copied all classes of the two themes into the new theme. We created new classes (`Server` and `ControlConn`) in the composed theme, and added the composed behaviour to them with aspects. Figure 6 illustrates the aspect that was created to compose the `EstablishConnection` and `Server` classes with merge integration and match-by-name. Here either of the integration techniques (merge or override) would provide the same results.

The aspect adds behaviour to the Server object. In the code, Advice 1 instantiates both the original classes when instantiation of

```
public aspect serverAspect {
  // introduce attributes
  private Server
    EstablishConnection_Server.thisServer;
  private
    EstablishConnection_Server Server.server1;
  private Session_Server Server.server2;
  private Socket Server.socket;

  // introduce the desired methods
  public static void Server.main(String[] args) {
  }
  public void Server.run() {
  }
  public void Server.dummySession(Socket socket) {
  }

  // Advice 1
  // instantiate the composing classes
  after() returning(Server server):
    call(public Server.new()) {
    server.server1 =
      new EstablishConnection_Server();
    server.server2 = new Session_Server();
    server.server1.thisServer = server;
  }

  // Advice 2
  // replace main method with an alternative
  void around(String[] t):
    execution(public static void
      Server.main(String[]))
    && args(t) {
    EstablishConnection_Server.main(t);
  }

  // Advice 3
  // replace server execution with an alternative
  void around(Server s):
    execution(public void Server.run())
    && this(s){
    s.server1.run();
  }

  // Advice 4
  // dummySession(Socket) provides a hook
  void around(Socket socket,
              EstablishConnection_Server s):
    call(public void
      EstablishConnection_Server.
      dummySession(Socket))
    && target(s) && args(socket) {
    s.thisServer.dummySession(socket);
  }

  // Advice 5
  // replace the hook with real functionality
  void around(Socket socket, Server s):
    call(public void
      Server.dummySession(Socket))
    && target(s) && args(socket) {
    s.server2.createSession(socket);
  }
}
```

Figure 6: An example aspect defined to integrate the `EstablishConnection` and `Server` classes.

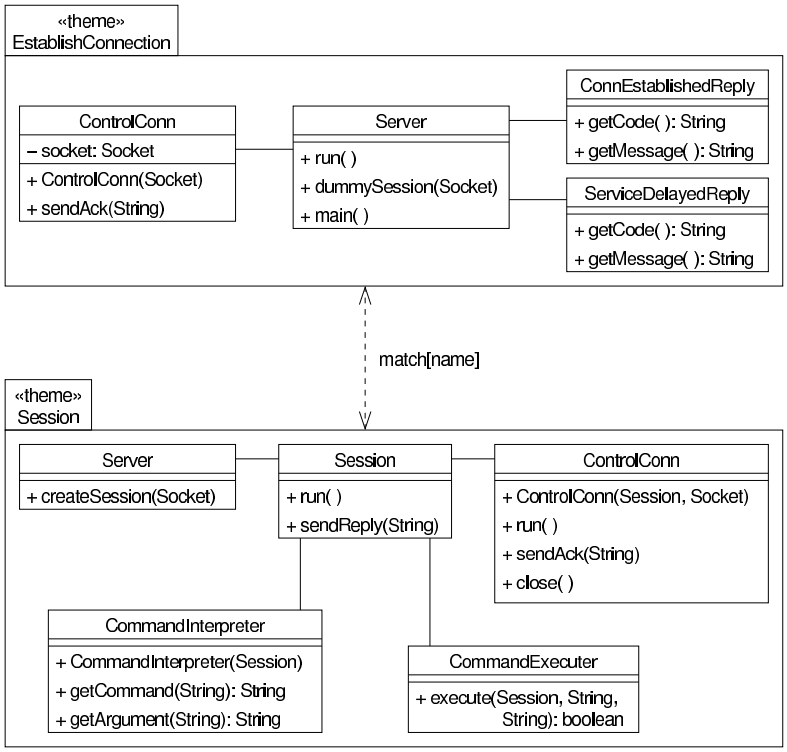

Figure 4: Two sample themes designed during the case study, one for the Establish Connection feature and the other for the Session feature. A simple composition relationship was defined between these themes.

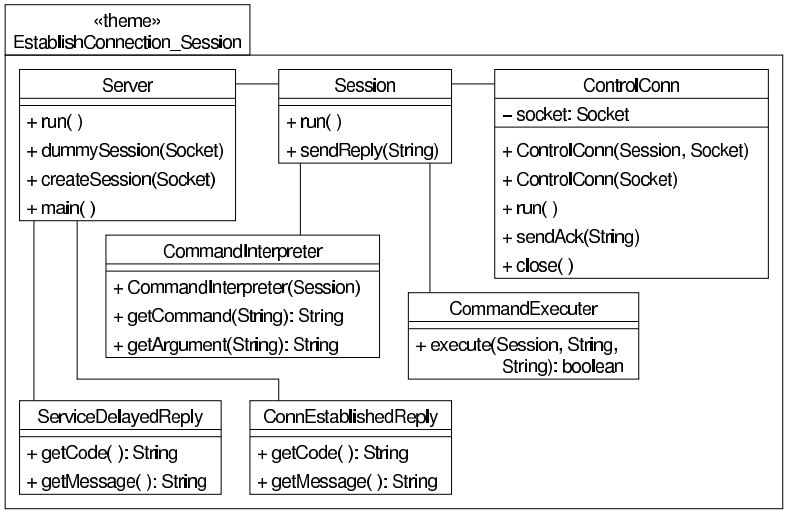

Figure 5: The theme resulting from the composition of the `EstablishConnection` and `Session` themes.

the composed class is attempted. This is a basic rule that we followed for composing any two (or more) classes that participate in any composition relationship. Advice 2 replaces the `main` method of the original `Server` class with the composed alternative. Advice 3 reroutes attempts to run the `Server` to the composed server object.

An explanation is needed for the presence of the `dummyMethod` in the aspect shown. As we discussed in Section 3.2.5, we initially missed a feature that was implicit in the informal specification for FTP, namely the need to provide separate sessions to clients as they establish connections. Thus, in the description of the `EstablishConnection` theme, no mention was made of passing control to a session upon setting up a connection with a client. Since this theme was oblivious to the existence of sessions, it was unable to pass control as needed. The `run` method of `EstablishConnection_Server` ran in a separate thread and as a result some mechanism was required to pass control to a session (by invoking `Session_Server`'s `createSession` method) from within the thread, whenever a client established a connection; two alternatives were available. As the first alternative, we introduced a dummy method `dummySession(Socket)` in `EstablishConnection_Server` to provide an explicit hook point inside the thread. The second alternative would be to capture the outgoing call (sending a reply message to the client) from within the thread via `after` advice. This advice would then invoke the `createSession` method. We chose to follow the first alternative because of drawbacks with the second as follows. To invoke the `createSession` method, it was necessary to capture the value of the `socket` field. While such capture could be accomplished by AspectJ's `set` pointcut, the Theme approach does not currently support a means for capturing the values of fields in compositions [7].

3.4 Evolution

In order to investigate the evolvability properties of the Theme approach, we evolved Version 1 of the system through a few additional versions along the identified feasible paths of the decision tree.

For Version 2 of the system, we added the user authentication protocol, requiring behavioural changes to several features of Version 1. A new theme for the `PASS` command theme was needed. The functionality of the `USER` theme needed to change to authenticate a user; this could be accomplished either through the modification of the existing theme, or through the addition of a new theme that overrides the old one. In this case, the authentication feature required such extensive modification to the `USER` theme of Version 1 that it made more sense to create a new `USER` theme to override the old one. All other command themes needed to test whether the session be authenticated, prior to invoking their main functionality, to ensure that only authenticated users can request some services. We selected this feature on the basis of its strongly crosscutting nature with respect to the existing features of Version 1.

To ensure that all command themes conform to the authentication protocol, two approaches could be followed: (i) adding the authentication-state check to all the command themes; or, (ii) providing the authentication-state check as a feature mapped into a new theme.

We tried the first approach by adding a `Password` class to every command theme so that it could check if the user were authenticated and thereby allow or disallow his/her request. But this approach required modifications of all command themes to incorporate the extra checking behaviour. The need for such widespread modification would indicate poor evolvability of systems developed under the approach. Therefore, we followed the second approach.

In Version 3 of the system, we added a new requirement to support passive mode with `PASV` command. In Version 4 of the system, we added a new feature to check for user account information with `ACCT` command. Both the features were selected because they crosscut other features of the system. In Version 5 of the system, we added support for the `HELP` command. This feature is a simpler one compared to the previous evolution steps, thereby assessing whether simple additive changes were made excessively complicated. All the three evolution steps have been attempted following the same approach as in the first evolution step (Version 2).

3.5 Path Feasibility and Lessons Learned

On exploring consequences of alternative decisions at different levels of the tree, we have traversed some paths that can be considered feasible, some that can be considered infeasible, and a few that remain topics for future study.

We have considered a feature-oriented decomposition of a specification into themes for the case study. As we discuss in Section 4, other decomposition techniques can be assessed for their feasibility and can be considered for future study. We have determined several feasible paths up to the 5th decision within the tree, as can be seen in Figure 3. In the final decision, two paths seem to be most feasible in our study. We have preferred the paths that consider some up-front planning in describing themes to the ones that do not consider any up-front planning at all; however, some of the unexplored nodes after the 6th decision cannot be completely ruled out as systems scale and evolution becomes more complex.

The two paths that we have considered most feasible in the study differ only in their treatment of how themes should be distributed among development teams. The results from each of these paths differed little.

In the course of exploring alternative decisions and investigating the consequences at different stages of software development, we have learned some lessons that can provide guidelines in applying the Theme approach in a more generalized manner. We have found that providing specific descriptions of themes is useful in supporting concurrent team development. Our lightweight approach to up-front planning may not work effectively in all systems, although an up-front plan can certainly help teams design themes in a more uniform manner, and making them more readily composable into a fully functioning system. Design-time composition has been an essential ingredient in our case study, but this may not be a general case; other decompositions into themes might exist that could avoid the need for design-time composition.

AspectJ does not function well as a tool for composition of independently developed, base-level implementation models. This is not surprising as it was created as a tool to weave explicitly crosscutting behaviour into base code.

4. DISCUSSION

This section discusses some important decisions that were made across the life cycle of the system development with the case study. Based on the study, we also present an initial evaluation of the Theme approach with respect to some key software properties like traceability, comprehensibility, and evolvability.

In exploring different paths in Decision 1 of the tree, we ignored Nodes 3 through k where different requirements engineering techniques could be investigated in deriving a structured specification. One reason was that, in our view, the results we obtained following feature-oriented decomposition would not vary much following those techniques, while requiring significantly more work. Realistically, they are approaches that would compete against Theme, and

not necessarily mesh nicely with it. But these approaches can be candidates for explorations in future study.

One node that we did not explore could make a difference in assessing the necessity of design-time composition verification of themes: Node 2.2, where we could map every requirement in the structured specification into themes. Our motivation behind not exploring that path was to determine if the system development was possible and feasible with Theme following a more lightweight approach. The feature-oriented decomposition seemed a lightweight one compared to decomposition into a traditional specification. This path can be explored in future study to find if this can provide a complete set of themes. We again note that we initially missed one theme via our lightweight approach, which was noticed during design-time composition. Whether spending more time up-front in decomposing themes can reduce the cost of verifying the system via design-time composition and incorporating changes accordingly.

We tried to approach the feature-oriented decomposition in a manner that can represent decomposition by any software development team, not biasing the decomposition results towards any particular design model. To justify our approach, we conducted a small survey of 5 graduate students in software engineering with varying industrial development experience. To minimize bias from leading questions, each of them was handed a copy of RFC 959 and was asked to decompose the system (Version 1) in a way that would support concurrent team development. None of them were told anything regarding the Theme approach or anything related to following any decomposition process. From their responses it was found that all of them suggested that different FTP commands should be distributed to concurrent teams with a few of them also suggesting that commands with similar functionality be given to one team. The tasks for integrating the commands or providing an interface for the core system to interact with users and performing different commands varied in their suggestions. But all their responses aligned closely with our derived set of requirements. Based on the survey, we can assume that our set of system features/requirements/tasks can be considered as a representative approach for decomposing this system.

In light of the case study, we found that the Theme approach could certainly improve some key software properties like traceability, comprehensibility, and evolvability, provided the development process follows a feasible route as discussed in Section 3.5. With requirements encapsulated into separate themes, each requirement can be traced through design and code. Moreover, dealing with a specific requirement would require understanding a specific theme; understanding a part of the design model and a package in the code can be sufficient to deal with a particular requirement. With the direct traceability and improved comprehensibility, evolvability is also improved as observed in the case study. Different evolution steps were performed via the same paths without non-localized modifications.

5. CONCLUSION

The Theme approach to AOSD considers full life cycle issues in moving from requirements through designs to implementations. To date, Theme has lacked evaluation of its core claims to improve key software properties like traceability, comprehensibility, evolvability, configurability, and concurrent development. This paper has provided a preliminary study in evaluating the Theme approach with respect to these claims. The paper presents a study on applying the methodology in developing and evolving a benchmark system, an FTP server. Since Theme is essentially a descriptive development methodology and not a prescriptive one, the study considered the many possible paths through the tree of development decisions in determining how (or whether) Theme can be applied in practice.

In traversing through a tree of development decisions, the study points out some infeasible paths that can be potential pitfalls, and explores some feasible paths that can provide directions for applying Theme to develop software in a more generalized manner. The study considered development decisions in all stages of the software life cycle. The results indicate that a feature-oriented decomposition of a system and then encapsulating each of the features into themes can lead to feasible solutions. The feasible paths in general have been reached through designing themes from explicit descriptions of the feature each of them captures. Attempting theme designs directly from a natural language description without their separate and explicit descriptions can lead to pitfalls. Lightweight up-front planning before designing themes in a concurrent development environment was more feasible compared to no up-front planning whatsoever. The study also suggests that design-time composition of themes developed by different teams is worthwhile, to validate the developed themes. The validation also helps composition of the themes implemented in parallel without any communication overhead. The study examined a process in implementing Theme composition with AspectJ; the development and different evolution steps support the feasibility and comprehensibility of the process, although AspectJ is not the ideal tool for supporting the symmetric model of AOSD.

6. ACKNOWLEDGEMENTS

We thank our anonymous survey participants for their assistance. This research was supported in part by a Discovery Grant from the Natural Sciences and Engineering Research Council.

7. REFERENCES

[1] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Int'l Conf. Softw. Eng.*, 2004.

[2] E. Baniassad, G. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Int'l Conf. Aspect-Oriented Softw. Dev.*, 2002.

[3] S. Clarke. *Composition of object-oriented software design models*. PhD thesis, Dublin City University, 2001.

[4] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 2002.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design. In *ACM Conf. Object-Oriented Progr. Lang. Syst. Appl.*, 1999.

[6] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *Int'l Conf. Softw. Eng.*, 2001.

[7] S. Clarke and R. Walker. Towards a standard design language for AOSD. In *Int'l Conf. Aspect-Oriented Softw. Dev.*, 2002.

[8] S. Clarke and R. Walker. Generic aspect-oriented design with Theme/UML. In R. Filman et al., editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[9] Y. Coady and G. Kiczales. A retroactive study of aspect evolution in operating system code. In *Int'l Conf. Aspect-Oriented Softw. Dev.*, 2003.

[10] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *Int'l Wkshp. Princip. Softw. Evol.*, 2001.

[11] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, 2002.

[12] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *ACM Conf. Object-Oriented Progr. Syst. Lang. Appl.*, 1993.

[13] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM T.J. Watson Research Center, 2002.

[14] M. Kersten and G. Murphy. Atlas: A case study. In *ACM Conf. Object-Oriented Progr. Syst. Lang. Appl.*, 1999.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Europ. Conf. Object-Oriented Progr.*, 2001.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Europ. Conf. Object-Oriented Progr.*, 1997. LNCS 1241.

[17] J. Kienzle and R. Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *Europ. Conf. Object-Oriented Progr.*, 2002.

[18] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating features in source code: An exploratory study. In *Int'l Conf. Softw. Eng.*, 2001.

[19] D. Parnas. On the criteria for decomposing systems into modules. *Commun. ACM*, 15(12), 1972.

[20] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Request for Comments 959, Internet Engineering Task Force, 1985.

[21] P. Tarr and H. Ossher. Hyper/J user and installation manual. Technical report, IBM T.J. Watson Research Center, 2000.

[22] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Int'l Conf. Softw. Eng.*, 1999.

[23] R. Walker, E. Baniassad, and G. Murphy. An initial assessment of aspect-oriented programming. In *Int'l Conf. Softw. Eng.*, 1999.

[24] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), 1971.