THE UNIVERSITY OF CALGARY

When Should Crosscutting Concerns Be of Concern in the Software Development Lifecycle?

by

Shafquat Mahmud

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

July, 2006

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "When Should Crosscutting Concerns Be of Concern in the Software Development Lifecycle?" submitted by Shafquat Mahmud in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Dr. Robert J. Walker, Supervisor
Department of Computer Science

Dr. Guenther Ruhe, Department Member
Department of Computer Science

Dr. Behrouz Far, Outside Member
Department of Electrical and Computer Engineering

Date

Abstract

Non-modularity of crosscutting concerns has been identified as a key impediment for object-oriented technology to address software evolution. Early and late aspect approaches, promoted through different AOSD models, address crosscutting concerns at different stages of the software lifecycle. The former is based on the concept of early identification and separation of crosscutting concerns, while the latter avoids the separation until design and addresses crosscutting concerns late in the lifecycle.

Both the approaches claim to achieve the software properties, traceability, comprehensibility, evolvability, and independent development, which are key to support evolution. Most AOSD techniques have focussed on means to apply either of the early or the late aspect approach. But no work to date has evaluated either of the concepts with respect to software evolution. As a result, the key question, whether either (or both) can provide a feasible solution to support the software properties important to evolution, remains unaddressed.

The thesis of the dissertation is that, addressing crosscutting concerns early in the software lifecycle fails to provide software evolvability and independent development; crosscutting concerns should be addressed late in the lifecycle to provide the software properties, which are key to evolution.

The thesis is supported by evaluating one AOSD technique, Theme, which has been promoted alternatively with both the approaches. A case study was performed to evaluate the two models of Theme with respect to the key software properties. The results of the study are analyzed and discussed in the dissertation to provide a general evaluation of when crosscutting concerns should be addressed in the software lifecycle, to better address software evolution.

Table of Contents

List of Figures

Acknowledgement

First of all, I would like to thank my supervisor, Robert J. Walker. I am indebted to you for your guidance and patience in steering me to the right path. I believe I have been able to learn some invaluable research skills from you that should help my endeavor for future research. It has been a real privilege working with you.

I would like to thank Kevin Viggers, Mohammad Minhaz, Shannon Jaegar, Jamal Siadat, Mark McIntyre, and Reid Holmes for sharing their valuable insights and experiences with me that have helped me better address the research topic. Also thanks to Rylan Cottrell, Bhavya Rawal, Brad Cossette, and Joseph Chang for providing a fun-filled working environment in the lab. I have thoroughly enjoyed the grad life with all of you around. I would also like to thank all my friends at Calgary, who have been like my family away from home.

I am grateful to my parents and my younger brother Shahriar, who have been the source of inspiration and motivation in every aspect of my life. Finally, I would like to express my appreciation to my wonderful wife, Ayreen. Thank you for being there to support me, when I needed you the most.

Dedication

To our first child.

We are eagerly awaiting your arrival.

Chapter 1

Introduction

Software aging is considered inevitable as we cannot construct a system to meet all future needs [65]. Successful software systems go through multiple evolution steps during their lifetime. As a result, support for evolution of software systems is a critical aspect of software engineering. Dijkstra [31] first coined the term "separation of concerns" early in the 1970's; a separate (modular) concern is expected to provide support for the software properties, which are valuable to evolution. A concern can be defined as any particular perspective of interest on a system, which may represent a task, a part of a system or a sub-system, a functional or non-functional feature of the system, a requirement or a group of requirements describing some system functionality, etc. [28]. Contemporary work in the 1970's by Parnas [64], argues that modularization can improve valuable software properties, specifically comprehensibility, changeability, and independent development. A modularized concern is expected to be simpler to understand and develop independently; it should also be easier to maintain and incorporate changes to a (modular) concern. Object-oriented technology attempts to separate and modularize concerns, but fails to address all concerns simultaneously. As a result, it fails to provide support for the software properties that are key to evolution. Recently, aspect-oriented technology has attempted to address these shortcomings. The question is whether it works.

1.1 Key Software Properties

The properties of "traceability", "comprehensibility", "independent development", and "evolvability", are identified as important for supporting software evolution. All these properties relate to the attribute, "modularization of concerns". These properties are of key interest in the thesis, and for the sake of brevity, we refer to them as the MC properties. Below we discuss what each of the properties means in software engineering perspective and how modularization of concerns can help achieving them.

Traceability refers to the property of being able to trace an entity of interest across the software lifecycle [67]. Typically, the software lifecycle consists of requirements definition (the degree of analysis or documentation varies among software models), design, implementation, deployment, and evolution. As a result, the support to trace a system requirement, or a concern across each of the lifecycle artifacts, is considered the traceability property of a software engineering approach.

The property of *comprehensibility* is typically considered as 'simplicity' to understand the design and implementation details of a software system, in order to modify or extend it [67]. Parnas [64] provides a more specific definition of the property, describing comprehensibility as the "*possibility to study the system one module at a time. The whole system can therefore be better designed because it is better understood*"; a module is considered as "*a responsibility assignment*" (e.g., any modular concern). Comprehensibility should facilitate a development team to deal with a particular concern easily either in the requirements, design, implementation, or the maintenance phase. As a result, development and evolution of the system should be

cost effective and the process should be less error-prone.

Based on the study by Belady and Lehman [16] on program evolution dynamics, Ciraci and Broek [21] have defined *evolvability* as, *"a system's ability to survive changes in its environment and requirements, in a cost effective way"*. Cost effective evolution refers to support for addition of new requirements without any modification to the existing system, and changes to the existing requirements in a localized manner that would not require invasive modification to any other part of the system; invasive modifications to different parts of a system can be difficult, costly, and error-prone to accomplish in case of large, complex systems [85, 25]. Support for software evolution, to include new requirements in a non-invasive manner (e.g., additive changes) and to modify existing requirements in a localized manner, can reduce the expensive evolution costs; as a result, evolvability is considered a significant and valuable software property.

Support for *independent development* is another important property for software engineering. It refers to the support for sub-teams to work independently without any communication overhead. In a practical development environment, development teams may need to work separately; communication overhead among development teams working at different times or at dispersed locations, can increase the development cost [18, 57].

By providing modularization of concerns, the MC properties should ideally be achieved. A modularized concern should be traceable across each of the lifecycle stages. Modularization should provide support for comprehensibility, evolvability, and independent development [64].

1.2 Object-oriented technology and the MC properties

Object-oriented (OO) technology is considered a standard means for industrial software development. The underlying object model provides OO with the ability to modularize different concerns. By providing modularization of concerns, OO should ideally provide support for the MC properties. However, not all concerns can be modularized for a particular decomposition of an OO system [50, 55, 40, 2]. In a complex system, certain concerns always crosscut any natural decomposition of the remainder of the system [51]. These concerns are referred to as *crosscutting concerns* that remain scattered across and tangled within multiple entities of the system [28]. *Scattering* refers to the non-localization property of a crosscutting concern, which is shared among multiple elements (in requirements, design, and implementation artifacts); and *tangling* refers to the intertwining of the details of multiple concerns within a single entity. Examples of some typical crosscutting concerns are non-functional requirements, such as persistence or security, and functional requirements implemented by collaborating objects, including design patterns.

Crosscutting concerns can impede standard OO technology from achieving the MC properties. Scattering and tangling of crosscutting concerns make it difficult to trace them across the software artifacts, from the requirements specification to the design and implementation units. Fig 1.1 represents an OO decomposition of a system specification into the design model. A rounded box represents an individual system requirement in the specification; each rectangular box represents an OO design unit. The dotted arrows represent the mapping (scattering and tangling) of the requirements in the specification, into the design units (classes). Details of a

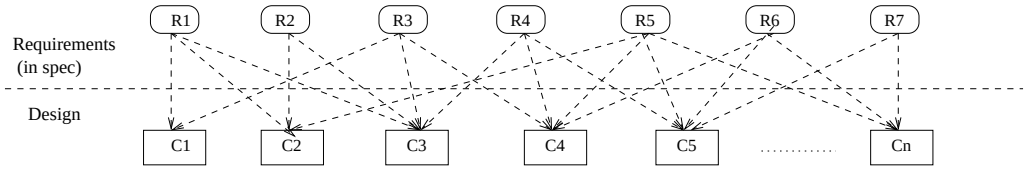

Figure 1.1: OO: Scattering and tangling of requirements among individual design units.

requirement in the specification, can get scattered across multiple design elements. As a result, implementation of a crosscutting requirement would get scattered across multiple implementation modules. Therefore, forward traceability of a (crosscutting) requirement or a concern, from the specification to the implementation, and backward traceability in the opposite direction, can be difficult. Moreover, each design (and implementation) unit typically needs to consider details of multiple requirements. The non-modularity of the requirements (crosscutting ones) can affect comprehensibility of the software system since, to comprehend a crosscutting concern, one might need to look into multiple elements (design and implementation artifacts) all across the system. An evolution task, relating to a single crosscutting concern, may as well require widespread modifications all across the system. Such invasive modifications concerning different parts of a system can be error-prone and difficult to incorporate, adversely affecting software evolution. Moreover, it may not

be a simple task to remove or plug in a crosscutting concern into the system, at any point of the development lifecycle. In the manner, crosscutting concerns can make standard OO vulnerable to the MC properties [51, 28].

1.3 Aspect-oriented techniques and the MC properties

Aspect-oriented software development (AOSD), a recently introduced software development paradigm, represents a set of technologies that attempt to cope with crosscutting concerns [51]. Based on how the crosscutting concerns are dealt with, AOSD technologies can broadly be categorized into two ideals: *asymmetric* AOSD considers separation of crosscutting concerns (termed, "aspects") from the base, in order to provide separate development of the base and crosscutting concerns; *symmetric* AOSD considers decomposition of the system specification into individual concerns, irrespective of the crosscutting behaviour [42].

With an aim to provide modularization of crosscutting concerns, asymmetric AOSD was initially introduced in the design and implementation phases of the software lifecycle. Crosscutting concerns are identified from an object-oriented design model and are explicitly separated out as aspects [51]; separation of crosscutting concerns from the base is expected to improve modularity of the system. Whether all possible crosscutting concerns can be identified and modularized from the design (and implementation) [81], or whether traceability of crosscutting concerns can be maintained from the requirements specification to the design and the implementation artifacts [72], have been the concerns with this approach. Expecting that identification and separation of crosscutting concerns from the system requirements

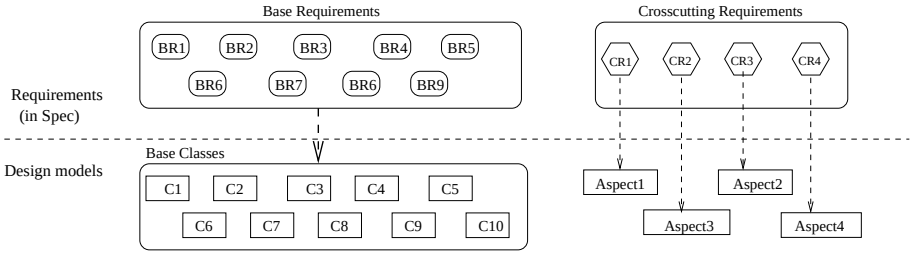

Figure 1.2: Early Aspects: Crosscutting concerns are mapped into separate design aspects.

can provide further improvements (better modularization of crosscutting concerns), a group of AOSD techniques have promoted the concept of early identification and separation of crosscutting concerns; these techniques are referred to as *early aspect* approaches. It is assumed that a thorough analysis of the system requirements can provide a comprehensive identification of all the crosscutting concerns in the system, and the early separation should also ensure their traceability across all the lifecycle artifacts [72].

Figure 1.2 represents an asymmetric decomposition of the system, according to the early aspect approach. A requirements engineering process, commonly referred to as aspect-oriented requirements engineering (AORE), identifies and separates out crosscutting concerns from the base requirements. The separation of the base and crosscutting requirements are shown in the figure. Each of the crosscutting requirements is mapped onto an individual design aspect, to be designed and implemented

separately. In most early aspect techniques, the base requirements are encapsulated into one design model (OO like). Figure 1.2 shows the mapping between the specification document and the design artifacts. The early identification and separation should ideally ensure that the separation between the base and crosscutting concerns is maintained throughout the lifecycle.

The ideal benefits of the early aspect approach are: (1) all crosscutting concerns can be identified early in the development lifecycle, and their modularization should ensure localization of the changes (to them) across the lifecycle artifacts; (2) crosscutting concerns can be added or removed without affecting the remainder of the system; (3) and, both crosscutting and base concerns can be understood independently and traced across different artifacts throughout the lifecycle. However, the issues with unanticipated evolution and independent development remain questionable since: unanticipated evolution can result in new crosscutting concerns or changes to the existing ones, either of which might require widespread restructuring of the existing base and aspect structures; aspects requiring explicit knowledge about the design details of the base, might also affect their independent development in practice.

Symmetric AOSD on the other hand, considers decomposition of the system into individual concerns, without explicitly considering identification and separation of the crosscutting ones. Each (symmetric) concern is encapsulated into an individual design unit (and its corresponding implementation module). Figure 1.3 represents a symmetric decomposition of the system. The rounded box represents an individual concern in the system specification that may refer to a single (or a group of) system requirement(s). The rectangular box represents a design model that encapsulates

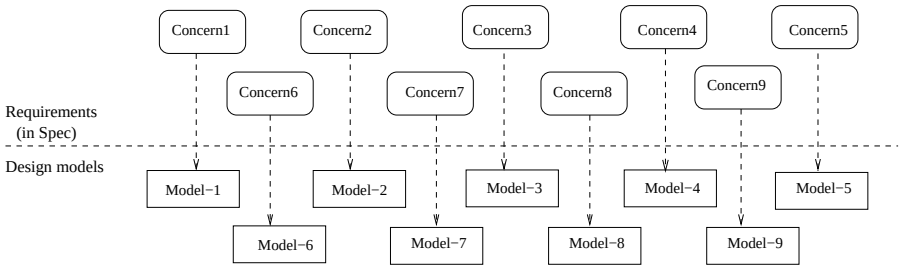

Figure 1.3: Symmetric AOSD: Symmetric concerns are mapped into separate symmetric models.

one concern. The arrow represents mapping of a concern (and the corresponding requirements) in the specification, into an individual design model. Each symmetric concern in the specification, design, or implementation artifact considers the embedded crosscutting behaviour *implicitly*, as the crosscutting requirements are not separated out here. Implicit consideration of crosscutting concerns refers to the fact that an individual software artifact treats a crosscutting or shared behaviour from its own perspective. Since each design (and the corresponding implementation) model considers the system from its own perspective without considering the explicit design details of other models, the views among different models can typically differ. These differences are essentially crosscutting concerns that need to be addressed in order to integrate the individual models into a complete system. Addressing crosscutting concerns late in the lifecycle (during integration), is referred to as the *late aspect* approach. Some AOSD techniques address crosscutting concerns late in the lifecycle

but do not promote a symmetric decomposition of concerns. However, in this thesis we mainly consider the late aspect approaches based on symmetric AOSD.

The ideal benefits of the late aspect approach, based on symmetric AOSD, are: (1) individual concerns are modularized and hence, can be traced and understood separately across the lifecycle artifacts; (2) a concern, not being dependent on any other part of the system, should be possible to develop independently; (3) and, changes to a concern should be localized to a single design and implementation model without requiring invasive modification to the rest of the system. New requirements should also be possible to include in an additive manner, and a concern in the specification along with the corresponding design and implementation model, should be (un)plugged without affecting the remainder of the system. However, practical applicability of the late aspect approach remains questionable since: individual design models treating crosscutting concerns implicitly might result in too many differences to be integrated (by a feasible and practicable means) into a fully functioning system. Changes to the crosscutting requirements might be problematic as well, since they may not be localized to a single design/implementation unit. As a result, non-invasive evolution of the system might not be achievable in practice.

In the next section, we present the broad research goal. The goal being too broad to address in a single thesis, we narrow it down to fit into the scope of a thesis. In Section 1.5, we present an overview of the previous AOSD research that relates to the broad research goal, before we describe the narrow research goal in Section 1.6.

1.4 The Broad Research Goal

Early and late aspects represent two distinct concepts for treating crosscutting concerns at different stages of the software lifecycle. The early aspect approach identifies and separates out crosscutting concerns (early) from the requirements specification, and maintains the separation throughout the lifecycle. By doing so, it claims to support the MC properties. The late aspect approach considers separation of concerns (from requirements) but not the crosscutting ones; crosscutting concerns are only addressed late in the lifecycle. In the process, the late aspect approach also claims to support the MC properties. Although both the approaches claim to address all the MC properties (key for software evolution), questions remain regarding their practical applicability and feasibility. Whether either (or both) of the approaches can provide a solution to address the valuable properties in practice, would one be better or more feasible over the other, or should one be preferred over the other in certain situations − remain unexplored. An evaluation of early and late aspect approaches regarding the unexplored issues, would be significant in addressing the general question of when in the software development lifecycle, crosscutting concerns should be addressed.

1.5 Overview of Related Work

Different AOSD techniques have been introduced based on the early aspect approach. Previous work [37, 6, 72, 58, 60, 61, 91, 6, 4, 71, 98, 33, 75] has mainly focused on the requirements analysis techniques to comprehensively identify and separate out crosscutting concerns from the system requirements. But, whether the early

separation of crosscutting concerns can be beneficial in achieving the MC properties, has not been evaluated. In fact, the effects (of separation) on software evolution remain unaddressed (or partially addressed) with most early aspect approaches; it is unclear whether evolution could be accomplished avoiding non-invasive modifications to the existing system, with an early aspect approach. Theme, an AOSD model, has been promoted via both early and late aspect approaches. The early aspect model of Theme [11] can be considered a notable early aspect approach that attempts to address software evolution in a non-invasive manner. The model claims to achieve all the MC properties.

Different AOSD models [76, 43, 44, 40, 73, 63, 54, 90, 32, 85], introduced based on the symmetric modelling of concerns, have promoted the late aspect approach. Theme [27, 26] has also been promoted based on the late aspect approach; it considers a symmetric decomposition of a system into separate concerns that can be developed separately, and it addresses crosscutting concerns late in the lifecycle. Some work [46, 95, 68], not explicitly based on symmetric AOSD, can also be considered as late aspect approaches, since they address crosscutting concerns late in the lifecycle. All the work promoting late aspects, has focused on design or implementation issues in applying the respective models; but no work to date has addressed the issue, whether late treatment of crosscutting concerns can be a useful means for addressing the MC properties, thereby supporting software evolution.

There has been some work [94, 12, 62, 49, 52, 29] on empirical evaluation of different AOSD approaches, mainly on different asymmetric models that separate out crosscutting concerns during the design and implementation. The results show that separation of crosscutting concerns during implementation can be beneficial on

certain occasions, and harmful on other situations.

Based on the evidence (in separating out crosscutting concerns from the design and implementation), the early aspect approaches assume that early separation of crosscutting concerns can be more beneficial and it should provide support for the MC properties. But no work has evaluated any early aspect approach with respect to the MC properties. Evaluation of either of the early or the late aspect approach, or any comparative study between them with respect to software evolution hence, remains unexplored.

1.6 The Narrow Research Goal

Although, it has been realized (mostly through AOSD work) that crosscutting concerns should be addressed in the software lifecycle, it is yet to be determined when in the lifecycle they should be considered, in order to better address software evolution. An evaluation of early and late aspect approaches with respect to the MC properties, and a comparison between the two concepts, remain significant software engineering research to be accomplished. However, such an evaluation on a comprehensive scale can be too broad an issue to address in a single thesis. Since different AOSD models have been promoted based on either of the approaches, it would require evaluation of all of them. Moreover, a particular model might suit a particular domain of systems more than the others. Evaluation of the different AOSD approaches with respect to the MC properties, considering all the constraints, is impractical to address in a single thesis, because of the associated cost and the time constraints.

In our work, we narrow the broad research question to fit into the scope of a

thesis. Our approach was to identify two AOSD techniques that could represent early and late aspect approaches in general, apply them in practice, and evaluate the MC properties across the software lifecycle; presumably, an evaluation of the two (representative) AOSD models on equal scales can provide an initial evaluation of the research question. The Theme model, promoted via both early and late aspect approaches, proved to be a suitable and reasonable AOSD technique for the evaluation process.

1.7 Theme as a representative AOSD technique

The early aspect model of Theme prescribes an explicit AORE process to identify and separate out crosscutting concerns from the system requirements specification; the separation is maintained throughout the lifecycle. Unlike many early aspect models, Theme does not prescribe an OO like decomposition of the base; rather it suggests for decomposition of the base into separate (base) concerns. The system specification is therefore decomposed into a set of base and crosscutting concerns. Each of the base and crosscutting concerns is mapped onto an individual design model. The Theme model also provides explicit mechanisms to design the individual models, and integrate them into a complete system. In the process, the early aspect model of Theme attempts to provide improved traceability and comprehensibility of individual system concerns (both base and crosscutting ones). Another significant characteristic of the early aspect model of Theme is its explicit attempt for software evolution in a non-invasive manner, the property that remains unaddressed or partially addressed in most early aspect approaches; in most early aspect models, it is unclear whether a

widespread restructuring of the base and the aspects can be avoided for unanticipated evolution of the system, but the early aspect model of Theme explicitly advocates for software evolution in an additive manner that should be able to avoid the possible restructuring.

The late aspect model of Theme, similar to a typical symmetric AOSD model, suggests for decomposition of a system into separate concerns, irrespective of the crosscutting behaviour. The model provides an explicit mechanism to model individual concerns separately, and integrate them later into a complete system. The differences among individual design models are addressed during the integration. By providing a symmetric modelling of concerns and considering crosscutting concerns late in the lifecycle, the late aspect model of Theme claims to achieve the MC properties.

Both the early and the late aspect models of Theme claim to achieve all the MC properties to support software evolution; many AOSD techniques address some of the properties, ignoring others. The early aspect model of Theme attempts to improve upon other early aspect techniques in achieving non-invasive evolution of software systems; the late aspect model of Theme represents a typical late aspect technique, based on symmetric AOSD, which claims to provide support for all the MC properties. An evaluation of the early aspect model of Theme thus can provide an evaluation of what best (with respect to software evolution) can be achieved by early aspect approaches in general, while an evaluation of the late aspect model of Theme should provide an evaluation of late aspect approaches (based on symmetric AOSD), in general. The early and the late aspect models of Theme, therefore, fare as suitable techniques to use as a basis for evaluating the underlying concepts of

early and late aspects.

The claim of the thesis is that the early aspect model of Theme fails to address independent development and evolvability, while the late aspect model of Theme fares to address all the MC properties to support software evolution.

1.8 Organization of the dissertation

Chapter 2 motivates the importance of early and late aspect approaches, and substantiates the thesis statement. It also provides a background on the Theme model, discussing its lifecycle. In Chapter 3, we provide an overview of the case study, conducted to validate the thesis statement. We discuss details of applications of the early and the late aspect models of Theme in Chapter 4 and Chapter 5, respectively. In Chapter 6, we analyze the results of the study, and discuss the overall evaluation process, its limitations, and its generalizability to address the broad research question. Chapter 7 discusses the related work more thoroughly, and we present the contributions and conclusion in Chapter 8.

Chapter 2

Motivation

Crosscutting concerns can be problematic for object-oriented technology to address the MC properties (traceability, comprehensibility, independent development, and evolvability). AOSD techniques attempt to deal with crosscutting concerns through two key approaches: early and late aspects. Both the approaches seem promising to address the software properties of interest. However, there can be issues of concerns regarding applicability and feasibility with either of them. With a view to evaluating early and late aspect approaches with respect to the MC properties, this thesis evaluates an AOSD model, Theme, which can be applied with both the approaches.

The goal of this chapter is to motivate the thesis statement by demonstrating the issues (regarding the MC properties) with standard OO, the early, and the late aspect models of Theme. To demonstrate the issues, we consider a partial banking system (as an example system) with the following informally described requirements specification.

The system must maintain different types of accounts for the customers. Each customer can have checking, savings, and mortgage accounts. A customer should be able to deposit, transfer, and withdraw money to and from the different accounts. Appropriate accounts should be debited or credited in the process. The system must calculate interest for customers appropriate to the different accounts. Charges will be computed for customer transactions, different for each of the accounts. Both interest and charges should be debited or credited to the appropriate accounts. The

system must maintain customer details, account ledger, and information of the bank branch that the customer is registered into. The system should be able to display a customer's account details, the branch information, and the account statements. The system should log transactions during transfer of amount, and account updates for charges and interest.

In the remainder of the chapter, we discuss how this system can be represented with the three different approaches (OO, early and late aspect models of Theme), and what can be the corresponding issues with respect to the MC properties for supporting software evolution.

2.1 Object-oriented representation

Figure 2.1 shows a UML class diagram for a representative object-oriented decomposition of the system (based on a similar design in [24]). Each of `Checking, Savings,` and `Mortgage` classes extends the superclass `Account`, and each of them encapsulates the respective behaviour for account updates, and calculations for interest and charges. `ChargeManager` and `InterestManager` classes provide methods to apply interest and charges into the general ledger. Transaction logs are handled by `LogFile` class.

In the design, not all concerns in the specification could be modularized; details of different concerns remain scattered across multiple design units (classes). For example, consider the functional concerns of "debit" and "credit". If we want to make changes to the concerns, we would need to make modifications into multiple different methods `(debitInterest(), creditInterest(), deductCharges(), credit(),`

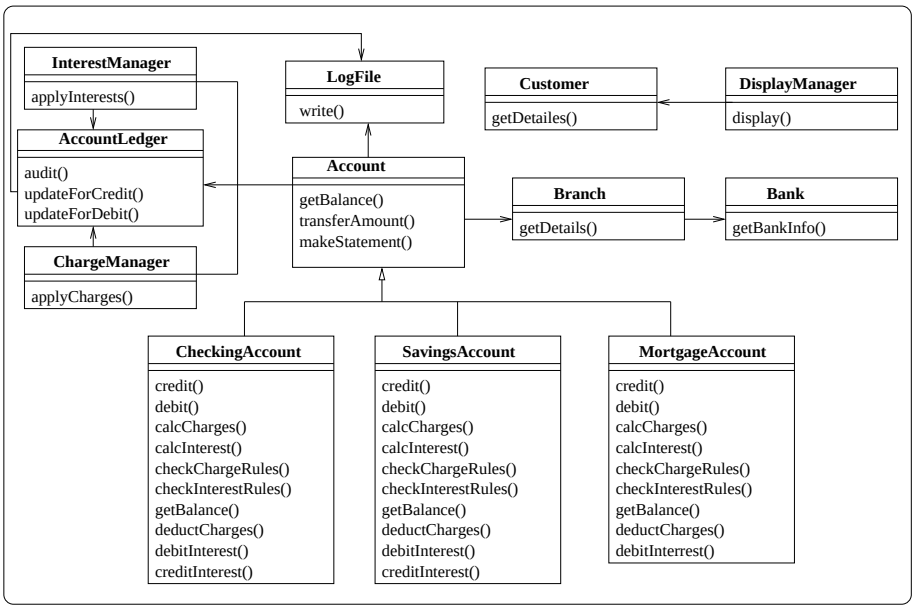

Figure 2.1: A representative decomposition of the system with OO

`and debit())` of `CheckingAccount, SavingsAccount, and MortgageAccount` classes, and also for operations (`updateForCharges() and updateForInterest()`) of `AccountLedger` class. Details of the non-functional requirement, "log transactions", are also scattered across the `Account` classes (`CheckingAccount, SavingsAccount, MortgageAccount`), within `transferAmount(), deductCharges(), debitInterest(), creditInterest()` operations, and also across `AccountLedger` class, within `updateForCredit() and updateForDebit()` operations. As a result, any change to a crosscutting concern would require modifications across different parts of the system.

This example only represents a sample specification for a partial banking system. Considering the complete specification of a large and complex banking system, changes to crosscutting concerns would typically require changes to a large number

of design elements and their implementations. With the crosscutting concerns not being modularized, it would be difficult to trace a concern or a requirement in the design (as well in the implementation). To deal with a single concern, a maintenance team might have to look into a number of classes all across the design model; this would affect comprehensibility. Evolution of an OO system would typically require invasive modifications all across the design model, adversely affecting the property of evolvability. As a result, OO technique would fail to address the MC properties, in order to support software evolution.

2.2 Theme representation of the system and issues

In this section we discuss how the example system can be represented with the early and the late aspect models of Theme; in doing so, we also provide a background of both the models. We then discuss the issues regarding the MC properties with either of the models.

2.2.1 Early aspect model of Theme

The early aspect model of Theme is based on the concept of early identification and separation of crosscutting concerns from the system requirements. It proposes Theme/Doc [10, 24], as an explicit process to identify and separate out crosscutting concerns from the system specification. To provide modularization of the base concerns, Theme/Doc also decomposes the base requirements into individual concerns. As a result, its product (after processing the system specification) are a set of base and crosscutting concerns. The base concerns are termed the “base themes” and

the crosscutting concerns, the "aspect themes". Each such concern is mapped into an individual design model in the design phase. A base design theme models the system from the perspective of the encapsulating base requirements only, without considering any crosscutting concern, while the aspect themes weave the crosscutting behaviour into the corresponding base themes. The Theme/Doc process takes a structured system requirements specification (SRS) and processes the requirements through the following steps to derive themes.

1. From each of the structured set of requirements, extract out the verbs as "actions" and qualify them as the *initial* themes.

2. Map each requirement into the relevant themes.

3. Unite/refine the initial themes to eliminate sharing of requirements among multiple themes.

4. Group similar themes together. Postpone/eliminate the unnecessary ones.

5. Rewrite requirements to eliminate sharing. Split/add requirements as necessary.

6. Determine the crosscutting requirements (the aspect themes) according to the following rules:

   (a) no rewriting or splitting of the requirement can isolate themes and remove sharing.

   (b) associating the requirement with a theme (dominant one) selects that theme as an aspect.

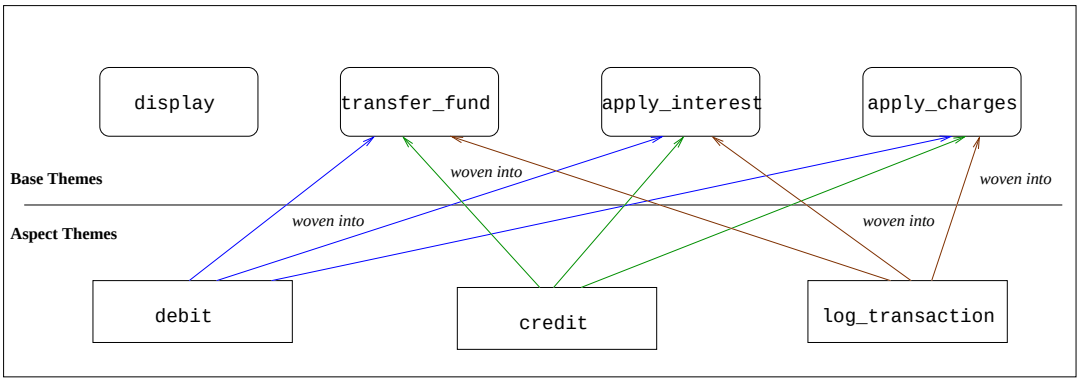

Figure 2.2: Aspect themes weave the crosscutting behaviour into the base themes

(c) the aspect theme is triggered from within multiple themes.

7. Iterate through steps 3 - 6 until all the aspect themes are derived and agreed upon, and a many to one mapping between the requirements and the themes is achieved.

Applying the Theme/Doc process on the example system specification, we can derive a number of base (`display, transfer_fund, apply_interest`, and `apply_charges`) and aspect (`debit, credit`, and `log_transactions`) themes. Figure 2.2 shows the relationships between the base and the aspect themes for the example system. Each of the base themes in the figure are supposed to model the encapsulating concern, being oblivious of the separated out crosscutting concerns. Aspect themes are supposed to weave the crosscutting behaviour into the corresponding base themes.

The Theme model proposes Theme/UML [25, 27, 23, 28] as an explicit mechanism to design and integrate the individual themes. It extends the standard UML to support separated, composable design themes. Each base theme is modelled as a stereotyped package that encapsulates the corresponding concern as an object-oriented design model; aspect themes are designed as UML packages with template parameters. Theme/UML also provides an explicit composition mechanism to integrate the individual design themes into a complete system. Below, we discuss design and integration of a couple of themes from the example system.

Figure 2.3 shows designs for two base themes `display` and `apply_interest`. Each theme only considers the system from the perspective of the encapsulating concern, and models it as an OO design model. The separated out crosscutting concerns are not considered on part of any of the base themes. `Display` theme considers only the details of displaying personal account information of the customer and information about the bank branch, s/he is registered into; `apply_interest` theme models the system encapsulating only the details of calculations and updates of accounts, corresponding to the accrued interest.

With each theme possessing different perspectives on the system, the perspectives need to be reconciled to form a complete system. Theme/UML provides a set of composition rules and relationships to provide integration of individual design themes. A composition relationship indicates which model entities in two themes correspond to each other and how they should be composed. The two most common composition relationships are *merge* and *override*. Merge indicates that the composed entity should be a true combination of the original entities. Override indicates

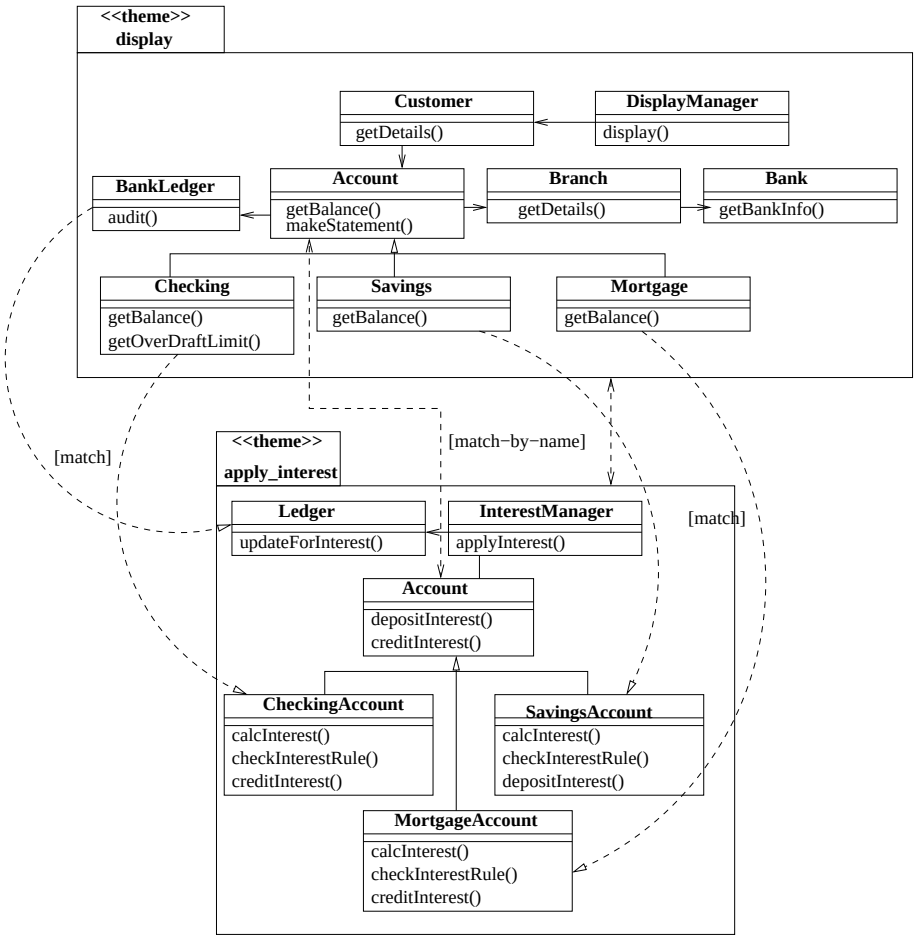

Figure 2.3: Design of two base themes and their relationships

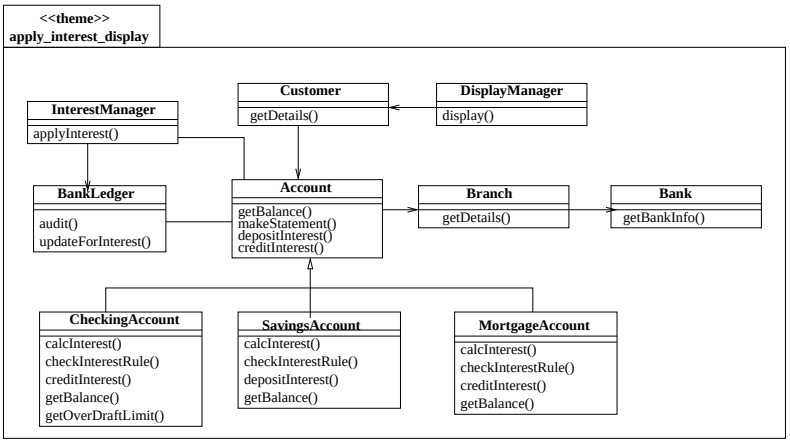

Figure 2.4: The resulting, composed theme.

that one should be discarded in favor of the other. Integration rules may be defined on composition relationships to avoid 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.

Figure 2.4 shows composition of the two themes in Figure 2.3 with merge integration and following the rules of match-by-name. A dotted arrow connecting the two 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

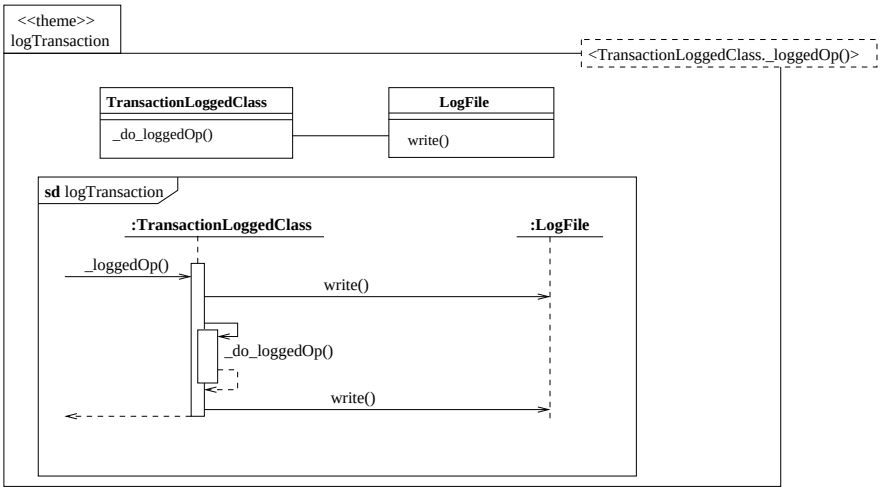

Figure 2.5: `log_transaction` aspect theme modelled with Theme/UML.

two classes (`Account` and `BankLedger`) are identical between the two themes. They participate in 'match-by-name' composition rule. The pairs (`CheckingAccount and Checking), (SavingsAccount and Savings)`, and `(MortgaegAccount and Mortgage`) are explicitly matched although the names do not match. Each such pair results in one composed class in the composed theme. The classes that do not participate in any composition relationship (`Customer, DisplayManager, Branch, Bank, and InterestManager`), are just copied into the composed theme.

Aspect themes on the other hand, are modelled with UML templates [24]. A template is defined as a parameterized UML model, used as the basis to generate other model elements using a "binding" dependency relationship [28]. A binding relationship defines arguments to replace each of the template parameters of the template model element. For an aspect theme, the template parameters are contained in a dot-

ted box which appears at the top right hand corner of the model. An aspect theme defines multiple *pattern classes*, which are the classes that are placeholders to be replaced by real class elements. All template parameters for the pattern classes are ordered in the dotted box, with each pattern class grouped within the "< >" brackets. Let us consider design of the aspect theme, `log_transactions`. Figure 2.5 shows a design for the aspect theme with one pattern class, `TransactionLoggedClass`, denoting that any class may be supplemented with log-transaction behaviour. The template parameter, `_loggedOp()`, defined for the pattern class, represents any transaction operation to be logged. A sequence diagram details the triggering of the crosscutting behaviour. The triggering behaviour (`_loggedOp()`), starts off the sequence by invoking `LogFile.write()` method, which performs the actual logging of the operation. Then the logged operation is executed with a call to `_do_loggedOp()`, which represents the invocation of the actual triggering operation. When the triggering operation completes, there is another `LogFile.write()`, and then the triggering operation returns to the base flow of execution. The aspect theme binds all the relevant operations of the base themes. Composition of the themes would result in a complete system.

Analyzing the MC properties

In this section, we analyze how the early aspect model of Theme addresses the MC properties, in developing the example system.

The model encapsulates each individual system concern into a separate theme (base or aspect). This should ideally ensure modularization of each individual concern and the corresponding requirements. A requirement should be directly traced

from the specification document into the design and the implementation. For example, let us consider two requirements from the example specification. One is, "*calculate charges for savings account*", and the second one is, "*debit checking account for a corresponding transaction*". The first requirement is encapsulated into the base theme, `apply_charges`, and the second requirement is modularized into the aspect theme, `debit`. Each requirement (base or crosscutting) in the specification thus, can be traced directly into the corresponding design theme, and its implementation. Such direct traceability should also support comprehensibility since, to deal with a particular requirement, one would only need to look into the corresponding theme. Changes to a requirement should also be possible in a localized manner. If we want modify a requirement regarding displaying of a customer information, we would only need to make localized changes into the theme, `display`.

However, an important concern prevails with all early aspect approaches: whether they can support evolution of the system in a non-invasive manner. An evolution step might introduce new requirements that crosscut existing decomposition of the system, requiring widespread invasive modifications. Let us consider an evolution of the system including a new feature to "provide Student account support", with the following requirements:

1. *"If a Student (a special customer), makes more than 12 transactions in the Checking account in a month, 3% of the transactions would be charged.*

2. *If the same student maintains CAD 5000 balance in the Savings account, 2.5% interest will be deposited into the account.*

3. *If a student is not charged for 3 months, the bank will increase the monthly*

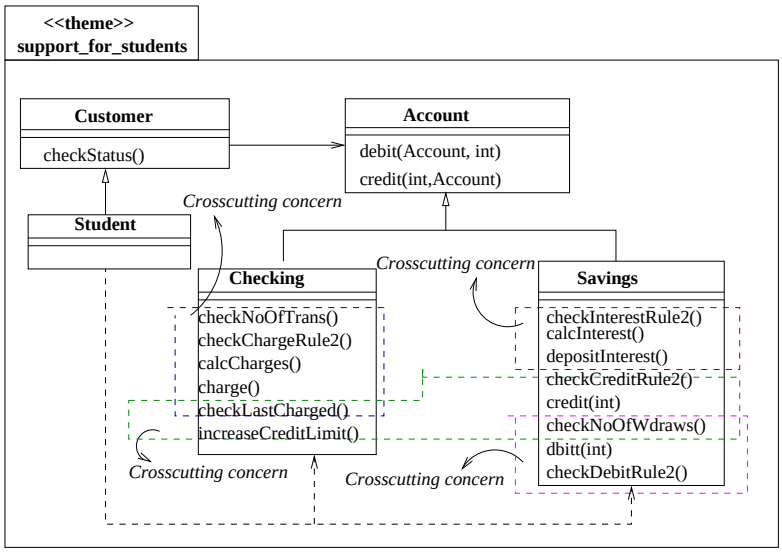

Figure 2.6: The new theme considers crosscutting concerns implicitly

Checking credit limit by 10%.

4. *If a student withdraws more than twice a month from the Savings account, s/he will not be allowed to credit anymore before s/he debits at least CAD 20. No debit is allowed on the last day of the month."*

According to the early aspect model of Theme, the new feature should be encapsulated into one or more new themes. Figure 2.6 shows a theme that encapsulates the new feature. The design theme can be considered a valid representation of the newly added feature; it encapsulates all the details for the new featrure and its integration with the existing system would result into a complete system. However, if we consider the complete design of the system after evolution, we would find that a number of crosscutting concerns have been addressed implicitly within the theme.

The highlighted parts in blue and red (in figure 2.6), indicate two new crosscutting concerns. These concerns crosscut the existing themes, `apply_interest` and `apply_charges`, along with the new theme, `support_for_students`. Had we processed all the requirements (existing and new) together during evolution, it would generate two new aspect themes to encapsulate the two separate crosscutting concerns. Since this would require invasive modifications to the existing system, as well as restructuring of the existing base and aspects, the early aspect model of Theme avoids consideration of the existing requirements in processing the new ones; the new requirements are encapsulated into new design themes only. But, as a result of doing so, the new theme has implicitly addressed these two crosscutting concerns. The theme also addresses another two crosscutting concerns (highlighted with green and pink dotted boxes) that should have been encapsulated into two existing aspect themes, `debit` and `credit`. By implicitly addressing the four crosscutting concerns from within the new theme, the clean separation of the base and crosscutting concerns has been affected, which is a contradiction to the early aspect approach itself. Based on the asymmetric AOSD, the early aspect approach follows the principle of maintaining the separation of the base and crosscutting concerns throughout the software lifecycle. But, to avoid invasive modifications to the existing system according to the early aspect model of Theme, here we have to address different crosscutting concerns implicitly (within the new theme) during evolution of the system, thereby breaking the asymmetric separation. Therefore, the early aspect model of Theme would have to break the asymmetric separation to provide evolvability, violating the key principle of the early aspect approach itself (e.g., to maintain the asymmetric separation); otherwise it would have to consider restructuring of the existing base

and crosscutting concerns through invasive modifications to different parts of the system, thereby failing to address evolvability.

The property of independent development can also be doubtful with the early aspect model of Theme. Independently developed themes can result in differences. But the early aspect model does not prescribe any means to address the differences among individual themes after their development. It is assumed that early analysis and decomposition of the system would be able to reconcile all differences, up-front. Whether differences arise in practice among independently developed themes, and whether the early aspect model of Theme suffices for a successful integration of independent theme into a complete system, should be interesting issues to investigate.

2.2.2 Late aspect model of Theme

The late aspect model of Theme considers a symmetric decomposition of the system into individual concerns. Each concern is considered as a general "theme" and is modelled as a separate object-oriented design model, with Theme/UML. Crosscutting concerns are not separated out, rather each theme considers a crosscutting concern from its own perspective, being oblivious of how any shared concern is considered by other themes (e.g. implicit consideration of crosscutting concerns). Each design theme, expected to be developed independently, can result in crosscutting differences; the differences are reconciled during integration of individual themes, e.g. the late treatment of crosscutting concerns.

Let us consider application of the late aspect model of Theme on the example system. The Theme model does not prescribe any specific means for symmetric decomposition of the system. A concern is considered as a task or a system feature,

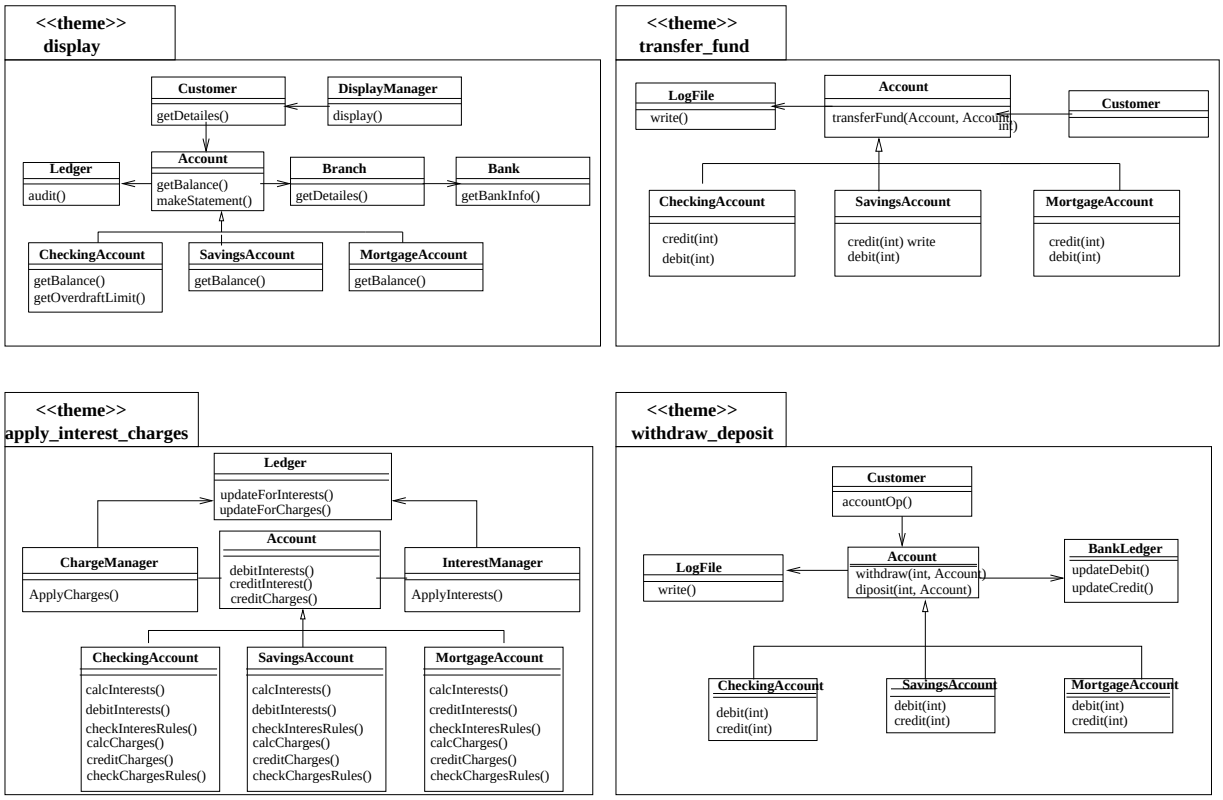

Figure 2.7: The late aspect model of Theme considers symmetric decomposition of the system.

irrespective of the crosscutting behaviour. Considering the example system specification, we can extract several features or tasks as themes; the system can be decomposed into the symmetric themes, `transfer_fund, display, withdraw_deposit`, and `apply_interest_charges`, corresponding to the features of transfer of funds between accounts, display of the customer information, withdraw and deposit of money, and application of interest and charges respectively. The individual themes are to be designed by Theme/UML and later integrated into the complete system.

Figure 2.7 shows designs for the themes with Theme/UML. Each theme models the system from its own perspective. With crosscutting concerns not being separated

out prior to the design, each design theme considers the embedded crosscutting concern from its respective view on the system. It can be observed that, behaviour for "debit" and "credit", which were separated out as crosscutting concerns in the early aspect model, are considered implicitly from within different themes (`transfer_fund, apply_interest_charges,` and `deposit_withdraw`). A composition of the themes with Theme/UML, is supposed to result in a complete system.

However, independent design themes can be expected to differ among their views on different entities of the system. For example, the theme, `apply_interest_charges` (in figure 2.7) may fail to consider the requirement, "*log the operations of applying interest and charges*", or it might have considered a different view of the log operation. A symmetric composition of the themes (with Theme/UML) would not be able to address such differences. These differences are essentially crosscutting concerns and are supposed to be addressed during integration of the themes. It is assumed that trivial and non-trivial crosscutting differences among individual themes can be addressed during integration, via the late aspect approach. However, whether this can be feasible in practice, remains an issue to explore.

Analysis of the MC properties

In this section, we provide an analysis of the MC properties for the late aspect model of Theme, with respect to the example system.

With symmetric decomposition of the system specification into individual concerns, we can achieve modularization of individual system concerns. Each requirement, corresponding to a particular concern, is mapped onto an individual design theme and its corresponding implementation. As a result, it would be possible to

directly trace any requirement in the specification, through the design and implementation. Modularization of concerns should also support comprehensibility; one should only need to look into the corresponding theme to deal with a particular requriement. Evolution of the system is supposed to be possible by adding new themes, without requiring any invasive modification to the existing system. For example, if we want to add the same set of requirements in evolving the system, as we did in the case of the early aspect model, we could add a similar design theme (as shown in Figure 2.6) to encapsulate the new requirements. Since the late aspect model of Theme does not follow an asymmetric AOSD, there should not be any issue of breaking the separation of the base and crosscutting concerns. We should be able to capture any concern (irrespective of whether there is any crosscutting behaviour embedded within or not) into a new theme and integrate it with the existing system.

The late aspect model of Theme seems promising to provide non-invasive evolution of the system. The model should also ideally support independent development of themes, since crosscutting differences are addressed (resolved) during integration of individual themes. However, whether we can achieve such non-invasive evolution and independent development in practice, remain unexplored.

Evolution of the system considering modification to crosscutting concerns can get difficult with the late aspect model. Let us consider modification of the system behaviour for "debit operation", requiring a different set of rules to debit an account. In the case of the early aspect model, this would have been localized to a single theme since, we separated out "debit operation" as a crosscutting concern (encapsulated within an aspect theme). But in the case of the late aspect model, we cannot make the localized changes. The late aspect model of Theme claims that evolution should

be possible in an additive manner. Whether we can accomplish changes to such a crosscutting concern by adding a new theme, would be an issue to explore. Moreover, whether late treatment of crosscutting concerns to integrate individual themes can be possible in a feasible manner, also remains unaddressed.

2.2.3 Summary

The motivational example shows how the early and the late aspect models of Theme, by addressing crosscutting concerns at different stages of the lifecycle, attempt to improve upon standard OO technology to address the MC properties. Although both the models claim to achieve all the MC properties, the early aspect approach proves susceptible to the properties of evolvability and independent development. The late aspect model on the other hand seems promising to address the software properties, important to evolution. However there can be concerns of its practical applicability and feasibility; crosscutting differences among individual themes may not be addressed in a feasible means after their designs, and changes to crosscutting requirements may not be possible in a non-invasive manner, as well. Applications of both the models in a practical development environment can be useful and valuable in evaluating them, with respect to the evolution issues.

In the remainder of the chapter, we discuss implementation issues with the Theme model. We also present an overview of the Theme lifecycle.

2.2.4 Implementation issues

Implementation of the integration of an aspect theme with the base themes that trigger it, has been prescribed with AspectJ [24]. However, no process or tool support

has been prescribed to implement integration of any two base themes, or the design themes in general (in the case of the late aspect model). We devised a mechanism to achieve the composition with either of "merge" or "override" integration rule using AspectJ. The process is described 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 is then defined to introduce the relevant attributes and operations into this new class, and to advise these introductions to invoke each of the original methods. Let us consider the composition relationship in Figure 2.3. Figure 2.8 illustrates the aspect that can compose the two `Account` classes with "merge" composition relationship and "match-by-name" rule.

In the code, Advice 1 instantiates both the original classes, when instantiation of 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 returns the `getBalance()` method of `apply_interest_charges.Account` class for any reference to the `Account.getBalance()` operation. Advice 3, Advice 4, and Advice 5 delegate the relevant method calls to the corresponding methods of the appropriate objects.

```
public aspect accountAspect {
 // introduce attributes
 private apply_interest_charges_Account Account.account1;
 private display_Account Account.account2;
 private Account apply_interest_charges_Account.thisAccount;
 private Account display_Account.thisAccount;

 // introduce the desired methods
 public int Account.getBalance(){
 }
 public void Account.makeStatement(){
 }
 public void Account.depositInterest(){
 }
 public void Account.creditInterest(){
 }

// Advice 1
 // instantiate the composing classes
 after() returning(Acount account): call(
    Account.new(..)){
        Account.account1 = new apply_interest_charges_Account();
        Account.account2 = new display_Account();
        account.account1.thisSession = account;
        account.account2.thisAccount = account;
  }

 // Advice 2
  int around(Account account): execution(
    public int Account.getBalance()) && target(account){
       return account.account1.getBalance();
  }

 // Advice 3
  void around(Account account): execution(
    public void Account.makeStatement()) && target(account){
        account.account1.makeStatement();
  }

 // Advice 4
  void around(Account account): execution(
    public void Account.depositInterest())
       && target(account){
          account.account2.depositInterest();
  }

 // Advice 5
  void around(Account account): execution(
    public void Account.creditInterest())
       && target(account){
          account.account2.creditInterest();
  }

}
```

Figure 2.8: An example aspect defined to integrate the `apply_interest_charges.Account` and `display.Account` classes.

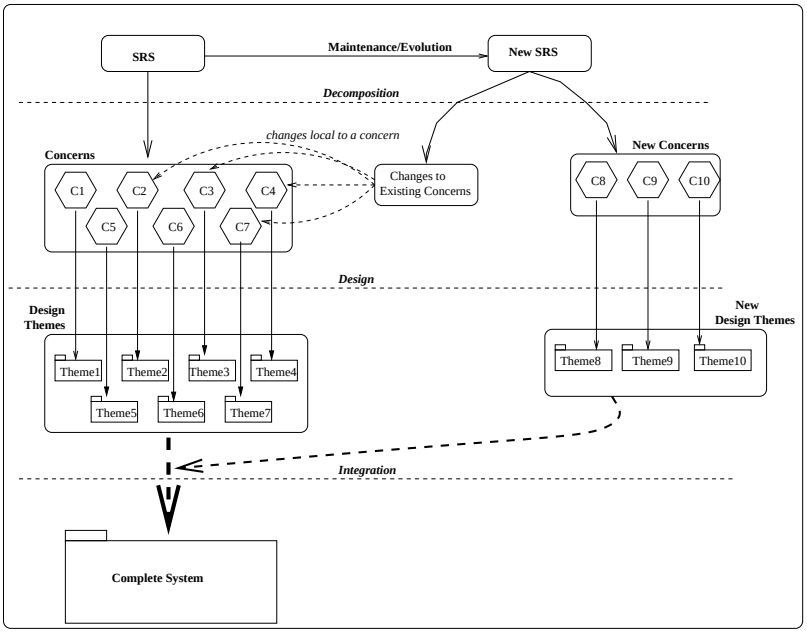

Figure 2.9: The life-cycle stages that involve the Theme model.

2.3 Lifecycle Issues with Theme

This section summarizes the lifecycle of Theme (with either of early or late aspects). Figure 2.9 shows the different stages of the lifecycle, the Theme model encloses. In a nutshell, the steps to develop a complete system from a system description with the Theme model would be to: decompose the specification document or the system description into a set of themes, design individual themes with Theme/UML, and integrate them according to the composition rules and relationships provided by Theme/UML, into a fully functioning system.

Theme has been promoted as a model to complement the standard software methodologies and processes, rather than to supplement them. In decomposing a system into a set of themes, the Theme model does not consider any of the requirements engineering practices [53] of requirements elicitation, analysis, negotiation,

documentation, or validation to derive a set of requirements. To complement software methodologies like Waterfall [56], Incremental [59], Spiral [17], Prototyping [78], or the agile models like Dynamic System Development Method (DSDM) [13], Crystal [74], and Agile Modelling (AM) [3], the development lifecycle for Theme would start from the system requirements specification (SRS) that is generated in case of each of the methodologies. The early aspect model of Theme applies its analysis tool, Theme/Doc, to separate out the base and the crosscutting concerns from the SRS as "base themes" and "aspect themes" respectively; the late aspect model of Theme on the other hand, decomposes the SRS into symmetric concerns (system functionality or features) that are termed as the general "themes". Various Iterative processes like eXtreme Programming (XP) [15, 14], and Scrum [1] do not produce any SRS in their lifecycle; user-stories (XP), or story-cards in the Product Backlogs (Scrum) are the sources of requirements for these processes. In applying the late aspect model of Theme to these processes, individual user-stories can be considered as separate themes. In the early aspect model of Theme, user-stories need to be processed by the analysis tool Theme/Doc to separate out base and aspect themes.

The derived set of themes are then designed with Theme/UML. The Theme model supports parallelism of development by permitting independent development of individual design themes, avoiding communication overhead; communication overhead can be considered disadvantageous, since it can increase the development cost exponentially [18, 57]. The issues of independent development here, raises a concern of whether independent themes should conform to a preplanned design, or whether they can be developed avoiding the up-front planning. Presumable the former approach would lead to less number of differences to resolve during integration, while

the latter one aims at avoiding the up-front cost by addressing resolution of differences during integration. Either might prove to be more feasible and cost effective over the other.

Theme/UML also provides a composition technique to integrate individual design themes. To implement a system applying Theme, one can directly implement the composed design, or implement individual design themes and integrate them later. Direct implementation of the composed design would be a waste of the whole approach, as it would result in an object-oriented implementation of the system as a whole. The practical decision therefore, would be to implement individual design themes and later integrate them according to the composition technique. Based on the amount of effort spent up-front in pre-planning, there would be number of differences among independently designed themes that must be resolved during composition.

Figure 2.9 also shows evolution in the lifecycle of Theme. An evolution step causes evolution of the SRS, or user-stories, or product backlogs. New requirements are mapped into new themes, and changes in existing requirements are mapped to the corresponding themes. In the early aspects model of theme, the newly added requirements may alter the structure of existing base and aspect themes. Since this would require widespread modification of the existing system, the Theme model suggests for mapping the new requirements to base and aspect themes only, not considering the existing ones. The newly derived themes and the existing themes that change (changes in existing requirements), then follow the next stages of development and integration.

Chapter 3

Evaluation: Scenario and Background

The goal of this thesis is to evaluate the early and the late aspect models of Theme, with respect to the MC properties; an evaluation of the Theme models is presumed to address the question of when in the lifecycle, crosscutting concerns should be addressed, in order to support software evolution. To evaluate the two models of Theme, we conducted a case study, where we applied both the models in the identical software development environment and investigated the software properties across the lifecycle. In this chapter we present an overview of the study, our selection of a suitable system for the evaluation process, and the criteria we considered in evaluating the models with respect to the MC properties. We discuss application of the early aspect model of Theme in Chapter 4, application of the late aspect model of Theme in Chapter 5, and we analyze and discuss the results of the study in Chapter 6.

3.1 Overview of the case study

In the study, we applied the Theme models in the development and evolution of a standard software system. We first developed an initial version of the system with either of the models; we investigated the feasibility and difficulties with respect to the properties of traceability, comprehensibility, and independent development at each stage of the development lifecycle. We then investigated the consequences of

multiple evolution steps on the base system, developed with either of the models, in order to investigate the software property of evolvability; each version was designed and implemented without consideration of any of the features for the next versions, to simulate the needs to accommodate unpredicted changes.

For the study, we considered the development and evolution of a File Transfer Protocol (FTP) server, which is described in an informally structured, natural language, requirements specification document (RFC 959) [66]. We selected FTP as the benchmark system [30] to investigate because it is well understood and small enough to analyze, but large enough to display difficulties involving the software properties of interest. FTP is a stateful protocol, involving the establishment of a control connection between a client and server for the exchange of commands and replies. 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.

3.2 The FTP study and our criteria for evaluation

The study began by developing the "Minimum Implementation" of an FTP server (as required by RFC 959) as the base system. In the development process, we considered consequences of following alternative decision paths along the lifecycle, especially in situations where the Theme model does not prescribe a specific path.

For example, in developing individual themes, we considered consequences of two alternative paths: one considering that individual themes conform to a pre-defined design architecture and the other one considering that their independent development with each theme having no prior knowledge about the design details of other themes. In the manner, we analyzed the path feasibilities and difficulties at each stage of the lifecycle. Later versions of the system added new features (from the remaining features described in RFC 959) to the base, or modified existing system behaviour. Each of the evolution steps incorporated changes that would typically crosscut a standard object-oriented system; the goal was to evaluate whether the Theme models could support software evolution to incorporate different crosscutting behaviour, avoiding invasive modifications to the existing system.

To evaluate the two models of Theme with respect to the MC properties, our criteria for success were as follows:

- A requirement in the specification can be traced to the design and the implementation, and vice versa.
- Any change to a particular requirement or a system feature should be localized to the corresponding theme only.
- Requirements can be added or removed solely through addition or removal of the corresponding themes, without altering other parts of the system.
- The occurrence of unanticipated changes can be dealt with in the same manner as anticipated ones.
- Alternative designs for an individual theme do not affect other designs themes.

Since the study was conducted by a single developer, it was difficult to analyze the property of independent development. To come round the problem and acquire results that can represent consequences of independent development of the system by distributed teams, we considered multiple alternative design decisions in developing each individual theme; the possible alternative design decisions for a theme is expected to simulate the different designs on the part of independent development teams.

In order to provide an evaluation of the two models of Theme on an equal basis, we considered their applications on the identical setting. For the development process with either of the models, we considered one software artifact as the starting point. RFC 959, containing the informal specification for FTP server, might have been considered the initial artifact to start with. However, the document contains the informal specification for a complete FTP server; for the base system (Version 1), we only considered the "Minimum implementation of FTP server". As a result, we derived a requirements specification (RFC 959min) for Version 1 of the system from the complete system specification. The derivation process involved copying of all the details from RFC 959 that correspond to the "Minimum Implementation of FTP" to produce an informal, natural language specification document. For the development process with either of the models, we considered RFC 959min as the initial artifact to start with.

In Chapter 4 and Chapter 5, we discuss applications of the early and the late aspect models of Theme respectively, and we analyze the results of the study in Chapter 6.

Chapter 4

Evaluation: Applying the early aspect model

This chapter describes application of the early aspect model of Theme in the development and evolution of an FTP server. In the development and evolution process, we explored and analyzed possible alternative decisions at each stage of the Theme lifecycle (described in Chapter 2). In Section 4.1, we discuss our approach in deriving a set of themes from the initial artifact of RFC 959min, we discuss the issues with development of the derived themes and their integration into a complete system in Section 4.2, and in Section 4.3, we discuss the consequences of the various evolution steps that we attempted on the base system.

4.1 Decomposition into themes

The early aspect model of Theme follows the Theme/Doc process to generate a set of base and aspect themes from the system specification. Theme/Doc takes a structured system requirements specification (SRS) as input, processes it through the steps (1 to 7) discussed in Chapter 2, and produces a set of base and aspect themes as output. Since our initial artifact for the development process, RFC 959min, contained an unstructured description of the requirements for the base system, we had to generate a structured SRS from the document in order to feed it to the Theme/Doc process. We name this additional step as "Step 0" of the process to derive themes. In the remainder of the section, we first discuss our approach of generating a structured

SRS and its informal verification as a valid representation of the base system. We then discuss the Theme/Doc steps we followed to derive a set of themes from the structured set of requirements.

4.1.1 Step 0: Derive a structured SRS

The specification document, RFC 959min, contained informal and unstructured descriptions of requirements grouped under relevant system features. To convert the document into a structured SRS, we analyzed each sentence under a particular feature and combined the relevant sentences to describe as a requirement. This also involved rewriting of sentences to describe a requirement. For example, the informal description of the system implicitly stated that the server would support multiple users, and requests from each user would be processed with respect to the particular (separate) connection between that user and the server. Since it was not separately stated in the document as an explicit requirement, we combined the bits and pieces regarding this concern and described this as a structured requirement (R51).

To verify the derived set of requirements as a correct representation of the system, we used the knowledge acquisition and requirements analysis technique of repertory grids [35]. This technique takes as input a series of elements relevant to the system domain and a set of constructs that can provide a comparison of the elements. It processes the inputs to generate a set of grids that show the relationships and differences among the domain variables; the grids can be represented in matrix, cluster, or map form to provide a graphical representation of the relationships. Repertory grid analysis can be a useful means to investigate understanding about a system's domain [80]. We used a repertory grid tool (WebGrid III) to derive grids for the

FTP server system; a sample grid in "map" representation is shown in Appendix A (Secion A.1.1). We then used the grids to investigate whether any of the structured set of requirements violates or contradicts any of the grids, by matching each of the requirements with the grid properties.

In the process (of Step 0), we generated a structured SRS containing a set of 51 requirements, which has been verified to be a valid representation of the system. Appendix A (Section A.1) describes the structured set of requirements.

4.1.2 Theme/Doc process to derive themes

Theme/Doc takes the structured SRS as input, produces an initial set of themes, and then refines the requirements and the themes throughout its requirements engineering process (via the 7 steps), until an acceptable (many to one) mapping from the requirements to the corresponding themes can be reached. This section discusses the Theme/Doc steps we followed to derive a set of base and aspect themes from the structured SRS.

According to the first step of the process, we extracted out the *verbs* (*actions*) from the set of requirements; with the actions being considered as the initial themes, the step resulted in 46 (initial) themes. Next (step 2), we mapped the requirements into relevant themes. The mapping resulted in numerous sharing among the requirements and the themes. We then followed the next steps (3 to 5) to reconcile the sharing among the requirements and the themes through grouping similar themes together, splitting and attaching relevant requirements, rewriting requirements, and postponing unnecessary themes. After multiple iterations through the steps, we could map all but 3 requirements into individual themes, where no require-

ment was shared among two themes (many-to-one mapping). The sharing of the 3 requirements among multiple themes could not be reconciled, rather they qualified as aspect themes (identified through Step 6). The 3 requirements, "*send reply via control connection*", "*send or receive files via data connection*", and "*process requests corresponding to the specific client-server connection*", all conformed to the (Theme/Doc) rules to qualify as `aspects` since, (1) no rewriting or splitting could remove the sharing; (2) attaching each requirement to an aspect theme made it the dominant one, and; (3) each one would be triggered multiple times by multiple base themes.

The output of the Theme/Doc process were 11 base and 3 aspect themes. The base themes (`user, port, mode, type, stru, retr, stor, noop, quit, listen_and_connect, and process_user_requests`) encapsulated the system features corresponding to different FTP-commands, listening to a port to connect users, and processing user-requests; the aspect themes (`send_reply_via_ControlConnection`, `send_receive_files`, and `process_under_separate_connection`) encapsulated the corresponding crosscutting requirements. We proceeded with the development and integration of the derived themes with Theme/UML, as we discuss in the following section.

4.2 Development and Integration of the derived themes

In developing the derived set of themes, we explored the consequences of two alternative paths: Path 1 considers development of themes all at one time, and Path 2 considers their independent development. In this section, we discuss feasibility of

each path in the design and integration of the derived set of themes.

4.2.1 Path 1: Themes developed all at one time

In exploring Path 1, we considered development of all themes together. Design and integration details for a number of base and aspect themes (along this path) are provided in Appendix B (Section B.1). The themes developed together, conformed to a defined communication protocol. With the explicit knowledge about the design details of individual base themes as well as other aspect themes, an aspect theme could comprehensively implement the corresponding crosscutting behaviour (to be triggered by other themes). The themes (considered to be developed all at one time) did not result in any difference or conflict among their individual perspectives on the system. As a result, they could successfully be integrated into a complete system (Version 1), according to the Theme/UML composition mechanism. Path 1 proved successful in developing a fully functioning base system.

4.2.2 Path 2: Independent development of individual themes

In investigating independent development of individual themes according to Path 2, we explored consequences of two sub-paths: one (Path 2.1) considering independent themes not sharing a common design architecture, and the other one (Path 2.2) considering a common design architecture for individual themes to conform to. A pre-defined design architecture along the second sub-path (Path 2.2) is expected to provide a uniform communication protocol for individual themes to interact with one another, with each theme knowing the explicit means to trigger any other theme. The reason for our exploration of the two decision paths here was to analyze and

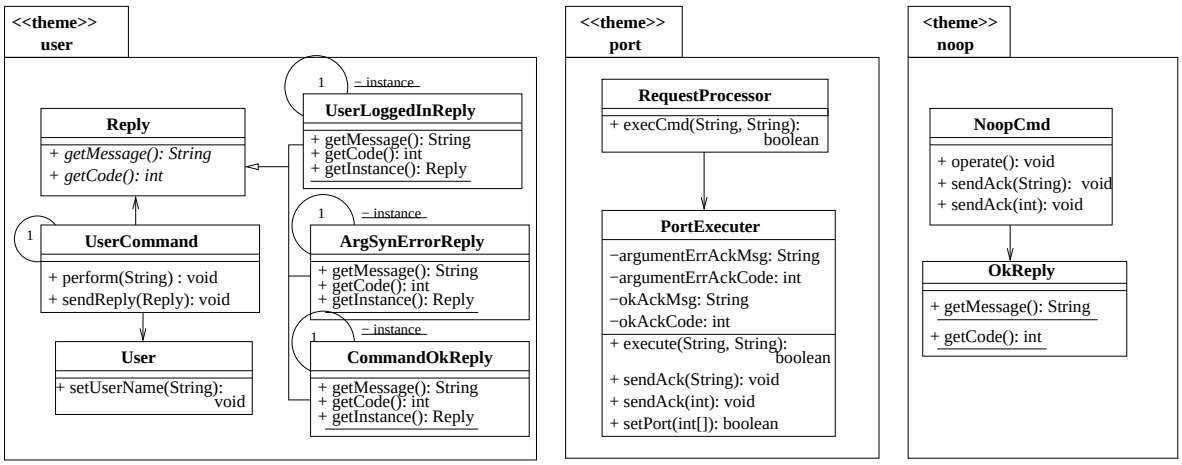

Figure 4.1: Alternative designs for the base themes

determine the feasible means for independent development of themes. In the remainder of the section, we first discuss development of themes considering the decision path (Path 2.1) of not pre-defining a design architecture for independent themes, and examine the feasibility of their integration into a complete system. Next, we examine feasibility of development and integration of themes along the second sub-path (Path 2.2), where independent themes are considered to follow a pre-defined design architecture.

Path 2.1: Independent themes do not follow a defined design architecture

To simulate independent development of individual themes avoiding any up-front effort in pre-planning a design architecture, we considered possible alternative designs for each theme. Figure 4.1 shows designs for three base themes, `user`, `port`, and `noop`; each of the three themes in the figure conforms to the encapsulated feature

description, and can be considered as a valid representation of the corresponding feature. `User` theme encapsulates the behaviour for the FTP-command, *user*. It expects to be communicated with by a call to `UserCommand.perform(String)` operation. It would store the user-name (passed as argument) and send a corresponding message. It should be noted here that none of the base themes consider any detail of the separated out crosscutting concerns. For example, the themes in Figure 4.1 do not consider the details of how a reply-message is sent to a user; the crosscutting behaviour (of sending reply messages) has been encapsulated into the aspect theme, `send_reply_via_controlConnection`. A request to send a message on part of any base theme (e.g, `port` in Figure 4.1), should trigger the crosscutting behaviour in the aspect theme.

`Port` theme encapsulates the behaviour for the FTP-command, *port*. The `PortCommand.execute(String, String)` operation determines the port information for a remote-user, stores the information, and requests for sending a corresponding message. The operation expects a command-name as the first argument passed; it returns 'false' for any irrelevant command (other than *port*), otherwise executes the FTP-command and returns 'true'. The `noop` theme encapsulates the behaviour for *noop* command. A call to `NoopCmd.operate()` operation would trigger the theme to execute the corresponding behaviour (of doing nothing and sending a message).

We explored alternative designs for individual themes, with a view to simulate their independent development. It was observed that each independent theme could be designed in multiple different ways. Since a theme need not possess explicit knowledge about the details of other design themes, no defined communication protocol

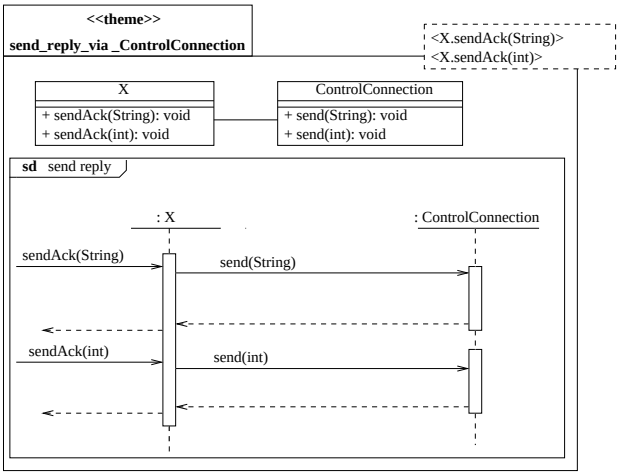

Figure 4.2: Aspect theme: `send_reply_via_ControlConnection`

could exist among individual themes to interact with one another. This made it difficult to design the aspect themes, as well as different base themes. We demonstrate the problem in the remainder of the section.

Figure 4.2 shows a design for the aspect theme, `send_reply_via_ControlConnection`. The theme expects to be triggered by a request to send a reply message to the user. The request is replaced by a corresponding *send* operation of an instance of ControlConnection class (shown by the sequence diagram). Let us consider integration of this aspect theme with the base themes in Figure 4.1. A mismatch would result in attempting to integrate `user` theme with this aspect theme. The template parameters (for the aspect theme), expecting to bind any operation of an instance of a class (X) with an argument of either of "string" or "integer" type, would support the crosscutting behaviour (of sending a reply-message) for `port` and `noop` themes in Figure 4.1, but not the `user` theme. Without knowing the details of `user` theme,

the aspect theme cannot presume the explicit means by which the base theme may trigger the crosscutting behaviour.

We also found it difficult to design the aspect theme `perform_under_separate_connection`, which encapsulates a more complex crosscutting behaviour, compared to the aspect theme in Figure 4.2. The `perform_under_separate_connection` theme is responsible to ensure that an FTP-command, requested by a user, would be processed with respect to the specific connection between that user and the server. To implement the crosscutting behaviour within the aspect theme, we realized the need to know the explicit means by which it could be triggered by different base themes. For example, let us consider `user` and `port` themes in Figure 4.1. Design for `user` theme suggests that an instance of `UserCommand` class, responsible to implement the behaviour for *user* command, should be shared across multiple user-connections. Storing of the 'user-name' into an instance of `User` class should however be specific to the corresponding connection between a user and the server. Design for `port` theme on the other hand, suggests that there should be an instance of `PortExecuter` class for each user-connection, since an instance of the class stores some information specific to a user. Therefore, these two base themes would trigger `perform_under_separate_connection` aspect theme for the corresponding crosscutting behaviour via different means. Without explicit knowledge about the design details of individual base themes, we found it impracticable to design the aspect themes.

We also found it difficult to design different base themes. For example, the base theme, `process_user_requests`, is supposed to implement the server behaviour to interpret requests from a user, and determine the requested FTP-command and

the passed argument value. For a valid FTP-command, the theme would need to communicate with a design theme that encapsulates the behaviour for that particular FTP-command. Considering an independent base theme would not possess explicit design details for other themes, we found that integration of the base themes resulted in communication mismatches. Appendix B (Section B.2) demonstrates a design for `process_user_requests` theme and discusses the communication differences in integrating the theme with any of the base themes in Figure 4.1.

Since the early aspect approach does not consider addressing the differences resulting from independent development of individual design themes, we found that the sub-path (Path 2.1) led to a failure in integrating individual themes into a complete system; the early aspect approach expects that the early separation of crosscutting concerns can resolve all differences up-front, but it did not hold in practice. Development of aspect themes proved impracticable without the explicit knowledge about individual base themes; design for individual base themes also proved insufficient without knowing the explicit means to communicate with other design themes. It would be necessary to address the differences among individual themes to integrate them into a complete functioning system.

Path 2.2: Independent themes follow a common design architecture

Expecting that a pre-defined design architecture can provide a uniform communication protocol for independent themes, we explored the decision path (Path 2.2) considering individual themes follow a common design. To simulate independent development of individual themes developed along this path, we explored alternative designs for each theme that conformed to the pre-defined design architecture.

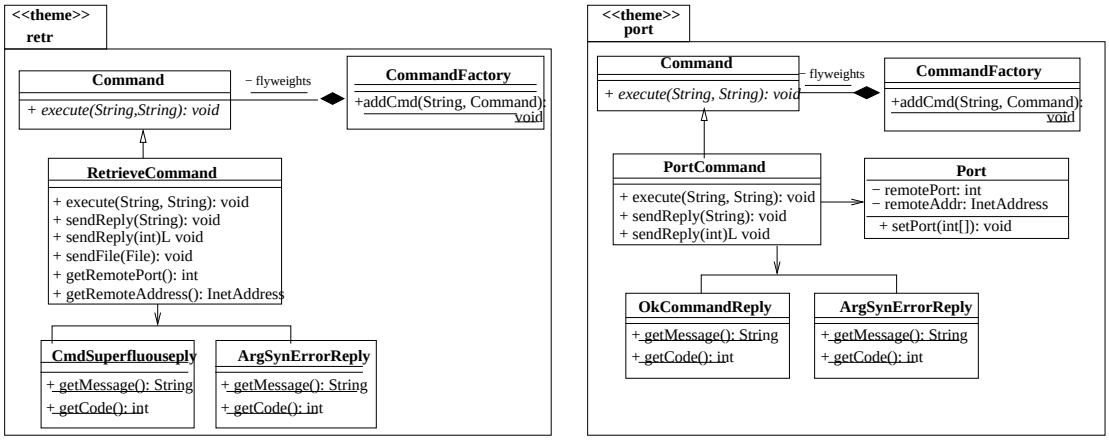

Figure 4.3: Independently developed themes follow a common design architecture

Figure 4.3 shows designs for two base themes, `retr` and `port` that conform to a pre-planned design architecture. It was decided that each theme, encapsulating an FTP-command, would be designed similar to Command design pattern [36]; a concrete `Command` class would implement the behaviour corresponding to a particular FTP-command. Since the themes would perform similar behaviour whenever triggered, each was to follow the design principle of Flyweight design pattern [36], to manage the references to the instance of a concrete `Command` class. This design definition sufficed for triggering of the crosscutting behaviour for "performing with respect to separate user-connections" in a uniform manner. It was also decided to trigger the other crosscutting behaviour (of transferring files and sending reply messages) by different base themes, in a uniform manner.

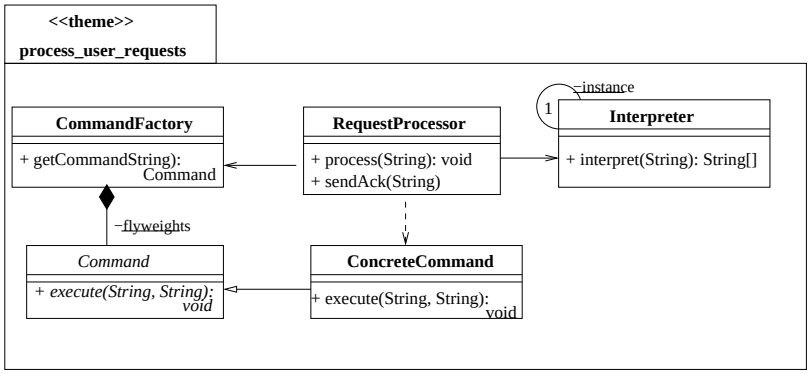

Figure 4.4: `Process_user_requests` theme

Following the pre-defined design architecture, the communication differences among individual themes could be resolved. Knowing how different base themes can trigger the crosscutting behaviour, the aspect themes could be designed accordingly. Figure 4.4 shows a design for the base theme, `process_user_requests`. With individual base themes following a common design architecture, this theme possessed the knowledge about how to trigger any relevant base theme. The theme in Figure 4.4 would be able to trigger any relevant base theme for a corresponding FTP-command (upon interpreting a user-request), by knowing about the concrete `Command` class from the Flyweight Factory (e.g. `CommandFactory` class); any theme (e.g, `retr` or `port` in Figure 4.3) encapsulating an FTP-command, would update `CommandFactory` class (Flyweight Factory) with the addCmd(String, Command) operation.

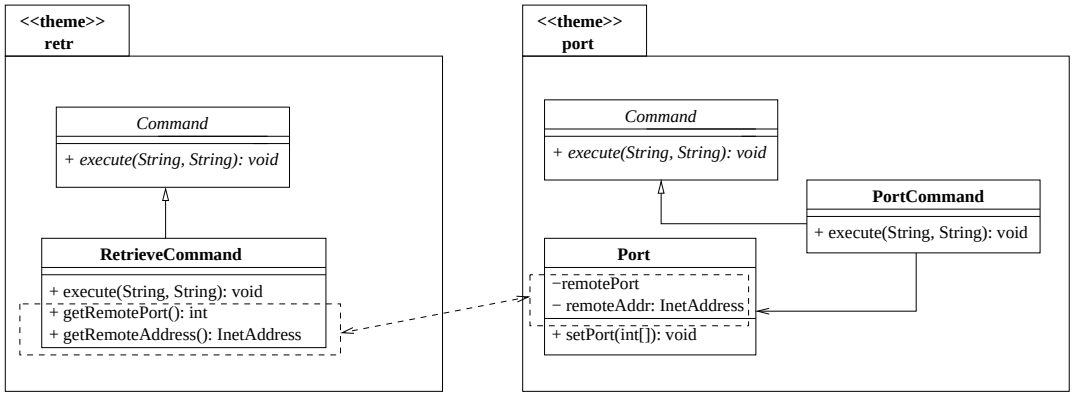

Figure 4.5: Independently developed themes resulting in differences

By providing a pre-defined design architecture, we could ensure a uniform communication protocol for individual themes to interact with. This sufficed for design of individual base and aspect themes; designs for the aspect themes in Appendix B (Section B.1) demonstrates how a pre-defined design architecture can suffice for successful addressing of the crosscutting behaviour. However, our explorations on alternative designs for individual themes conforming to the pre-planned design, showed that independently developed themes could still result in differences among their views. We present a scenario below.

Figure 4.5 extracts out parts of the design themes, `retr` and `port` (in Figure 4.3), that conformed to the pre-defined design architecture. The highlighted parts connected by a dotted arrow, show the differences in views of the two themes. In order to send a file to a remote-user, `retr` theme needs to access port information

about the particular user; the port information are set by some other theme (e.g., `port`). Composition of the two themes (`retr` and `port`) is supposed to provide a correct correspondence here. However, their composition with Theme/UML cannot resolve the differences. The operations, `getRemotePort()` and `getRemoteAddress()` of `RetrieveCommand` class (in `retr` theme) correspond to the attributes, `remotePort` and `remoteAddress` in the `Port` class (within `port` theme) respectively. But the two classes (`RetrieveCommand` and `Port`) cannot participate in any composition rule or relationship during composition of the two themes with Theme/UML. As a result, the differences would remain unresolved after the integration. Such trivial difference can result from independent development of themes and may not be predicted or resolved earlier.

In exploring alternative designs for individual themes, we observed that independently developed themes could result in differences and conflicts (similar to the one shown in Figure 4.5), even after following a pre-defined design architecture. Such differences are essentially crosscutting concerns, but they could not be identified earlier in the lifecycle and thus could not be dealt with by the pre-planned design architecture. We could identify the crosscutting differences among individual themes only during their composition, since these differences arise from low level design details. As a result, only the early addressing of crosscutting concerns did not suffice for development of a complete functional system; we would also need to address crosscutting concerns during integration of independently developed themes, in order to integrate individual themes into a complete system.

Summarizing the findings in exploring Path 2, considering independent development of themes (either through Path 2.1 or Path 2.2), we found that the early

aspect model of Theme would not suffice for a successful integration of individual themes, unless considering addressing of crosscutting concerns during integration; e.g., it would need to incorporate support of the late aspect approach, to result into a complete functioning system. Therefore Path 1, considering development of themes all at one time, is the only feasible development path with the early aspect model of Theme. We proceeded with evolution of the system developed along this path.

4.3 Evolution

This section discusses the evolution steps we attempted on the base system, developed through the only feasible path (Path 1). According to the lifecycle of the Theme model, as discussed in Chapter 2, each evolution of the system consists of 3 phases: 1) analysis of the new set of requirements to derive new base and aspect themes, 2) development of the new themes, and 3) their integration with the existing system. Since derivation of themes (via Theme/Doc) follows the same process across different evolution steps, we only discuss it for the first evolution step. In the remainder of the section, we discuss all the three phases of the first evolution step (Version 2 of the system) and discuss only the development and integration issues for the next evolution steps (Version 3 to Version 5).

4.3.1 Version 2

For the first evolution step (Version 2), we considered inclusion of a new feature, "user-password authentication protocol". We added the requirements corresponding to the new feature into the specification document (the structured set of require-

ments are presented in Appendix A, Section A.2). The added requirements referred to support for *pass* command, new behaviour in the case of *user* command, and verification of password-authentication for the FTP-commands. In the remainder of the section, we discuss the evolution step considering all the 3 phases: derivation of new themes from the requirements, their development, and their integration into a complete system.

In deriving the new set of themes for an evolution step, the Theme model suggests processing only the new set of requirements with Theme/Doc, ignoring the existing ones, to avoid possible invasive modifications to the existing system. Similar to our approach in deriving themes for the base system, we first followed "Step 0" to structure the new set of requirements. We then followed the Theme/Doc steps (1 to 7) to process the requirements and derive new themes.

Following the initial steps of the Theme/Doc process, we derived three (initial) themes, `new_user, pass`, and `verify_authentication`. At this point we could consider two alternative means to proceed through the decomposition process. The new requirements, corresponding to the modified behaviour in the case of *user* command, could be considered as 'change-requests' for the existing system, and therefore be mapped into the existing base theme, `user`. This also supports the property of 'localization of changes' for a particular concern. However, the requirements could also be considered as part of the new feature; mapping them into an existing theme might affect the (un)pluggability property (for the new feature). To avoid this, we could encapsulate the changed behaviour for *user* command into a new theme (`new_user`), to be merged with the existing behaviour later during integration. We proceeded with both the means to investigate further consequences.

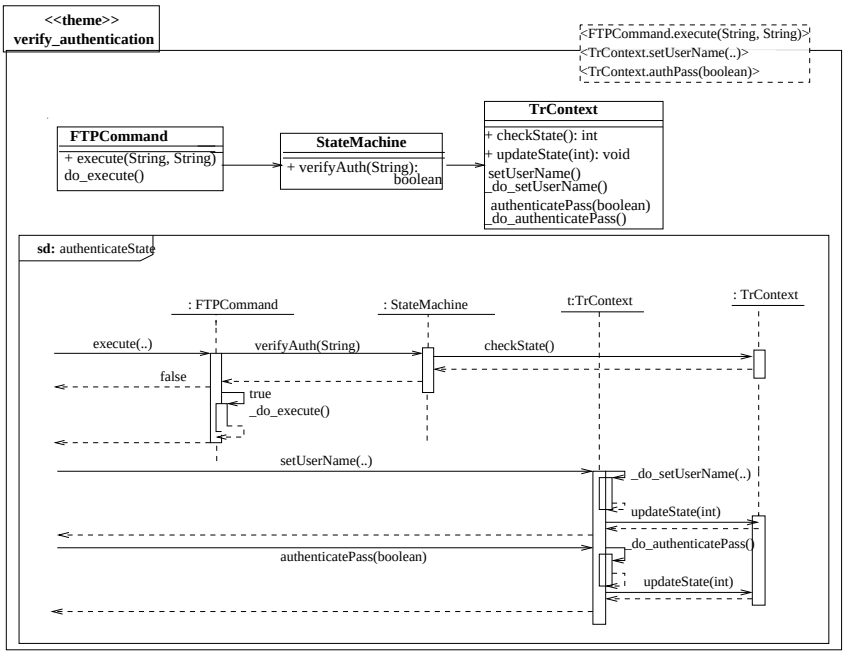

Figure 4.6: Aspect theme: `verify_authentication`

We iterated through the remaining steps of the Theme/Doc process and derived the new set of base and aspect themes. Appendix A (Section A.2.2) shows the initial and the final mapping among the new set of requirements and the themes. In the process, we derived two new base themes, `new_user` (or changes into `user` theme) and `pass`, and an aspect theme, `verify_authentication`. We then proceeded with development of the derived themes and their integration with the existing system.

We designed the base themes, `new_user` and `pass`, similar to the ones in the original version. Figure 4.6 shows design for the aspect theme, `verify_authentication`, which encapsulates the crosscutting behaviour to verify the machine state before allowing execution of the FTP-commands. The template parameter, `FTPCommand.execute(..)`, binds a corresponding operation of a theme that encapsulates an FTP-command. A call to execute an FTP-command triggers the crosscutting authentication verification

behaviour; the `verifyAuth()` operation (in `StateMachine` class) is invoked, which checks the state of the machine to (dis)allow execution of the corresponding FTP-command. The machine can be in three states: State 0, State 1, and State 2; State 0 represents the initial state after log-in, where the system expects a request for *user* command to proceed; State 1 represents the state after the system has accepted a *user* command and expects the *pass* command to authenticate a user, and; State 2 represents an authenticated machine state, where the system would allow all supported FTP-commands from a user. The aspect theme (`verify_authentication`) verifies the machine state before approving execution of an FTP-command. The theme also updates the machine state right after a user logs-in with *user* command (represented by the triggering operation, `TrContext.setUserName(..)`), and also when a user is authenticated with a valid request through *pass* command (represented by the triggering operation, `TrContext.authenticatePass(..)`).

We attempted to integrate the new themes with the existing system, according to Theme/UML composition rules and relationships. although the new themes encapsulated all the details corresponding to the new feature, the integration attempt resulted in an incomplete system. The new themes `new_user` (or changes into the theme, `user`) and `pass` were consistent with the existing design of the base system (e.g., can be considered as anticipated changes); as a result, their integration with the existing system was trouble-free. But the existing system (Version 1) did not have any knowledge of a state machine. In the original system, `perform_under_separate_connection` aspect theme was responsible to ensure that access of all information for a particular user would be processed corresponding to the particular connection between the user and the server. With no knowledge about the

state machine, the aspect theme did not suffice to support processing of state verification (for the new aspect theme) with respect to the corresponding user-connection. Not considering the existing requirements while processing the new ones, this behaviour went missing in the analysis process with Theme/Doc. Had we processed all the requirements together (during the evolution step) with Theme/Doc, we would have found that the existing crosscutting requirement R 51 (stating that access to all information corresponding to a user should be processed with respect to the particular connection with that user) also crosscuts the new theme. Modifications to the encapsulating aspect theme (`process_under_separate_connection`) to support the new requirement would have resulted in a successful integration. But the early aspect model of Theme avoids such an approach of taking the existing system requirements into consideration, as it might lead to widespread invasive modifications to the existing system. Following the Theme approach, we missed certain system behaviour and the evolution step resulted in an incomplete system.

Let us suppose that a better requirements engineering approach could find the missing requirement in processing the new feature; we might also have missed this requirement in our elicitation and analysis of the new requirements (corresponding to the feature). To investigate how the Theme model would fare had we identified the missing requirement (R 62: "*the state verification should be processed with respect to the specific user-connection*"), we included the requirement in the description of the new feature, and processed with Theme/Doc. This resulted in similar three themes, with this requirement (R 62) being mapped into the aspect theme, `verify_authentication`.

Figure 4.7 shows a design for the aspect theme including this requirement. The

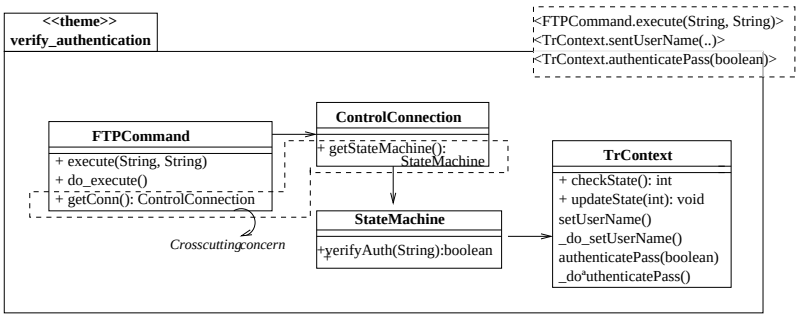

Figure 4.7: The theme implicitly addresses another crosscutting concern.

highlighted part shows that this theme has implicitly addressed an already separated out crosscutting concern. If all the requirements were processed together, the crosscutting concern would have been encapsulated into the existing aspect theme, `process_under_separate_connection`. Implicit consideration by this theme of the crosscutting concern, breaks the clean separation of the system. This contradicts a key principle of the early aspect approach, e.g., to maintain the clean separation of the base and crosscutting concerns throughout the software lifecycle.

Therefore the first evolution step proved to be an unsuccessful one; the early aspect approach of Theme, in an attempt to avoid invasive modification of the existing system, either resulted in an incomplete system failing to identify a system requirement, or had to break the asymmetric separation of the system by implicitly considering a crosscutting concern. We however, explored more evolution steps to further investigate the evolvability property of the early aspect model of Theme.

4.3.2 Version 3

For version 3 of the system, we included the system feature corresponding to the FTP-command, *rein* (re-initialize user). This feature requires to check for any file transfer in progress. If so, the server would wait for the transfer to complete, close the data connection, flush the user information stored, and restore the default state, which corresponds to the initial state right after the establishment of a user-connection. In this section, we discuss our design for the new theme(s) to address the added feature and the consequences of the integration with the existing system.

Processing of the new requirements with Theme/Doc resulted in one new base theme, `rein`. Figure 4.8 shows a design for the theme, which captures the details corresponding to the new feature. Integration of this theme with the existing system resulted in a complete functioning system (Version 3). Looking into the design of the theme in Figure 4.8, one would find it reasonable, as it encapsulates the corresponding requirements. However, considering the design of the complete system, we found that this design theme has implicitly addressed several crosscutting concerns. Had we considered all the requirements together, we would have found three crosscutting concerns being implicitly addressed within this theme. Crosscutting concern 1 in the figure (relating to flushing of user information) crosscuts the `user` theme (in Version 1, or the `new_user` theme in Version 2), the `verify_authentication` theme (in Version 2), and the `rein` theme (in Version 3); Crosscutting concern 2 (relating to checking for an ongoing file transfer), crosscuts the `send_receive_files` theme (in Version 1) along with this new theme; and Crosscutting concern 3 (relating to restoration of the machine state), crosscuts the `verify_authentication` theme (in

Version 2) along with this newly added theme. Since separation of these crosscutting concerns would lead to restructuring of the existing base and crosscutting concerns, as well as invasive modifications to different parts of the existing system, the early aspect model of Theme avoids consideration of the existing requirements during an evolution step; the new requirements are encapsulated into new design themes to achieve evolution in an additive manner.

With a view to achieving non-invasive evolution according to the early aspect model of Theme, the new theme (`rein`) has considered multiple crosscutting concerns implicitly. Such implicit consideration of crosscutting concerns on the part of `rein` theme here, breaks the clean separation of the system. This evolution step has also resulted in a system that does not represent an asymmetric model according to the early aspect approach, as the clean separation between the base and the crosscutting concerns has not been maintained. We can hence consider this evolution step an unsuccessful attempt as well.

4.3.3 Version 4

In the next evolution step, we considered inclusion of the feature corresponding to the FTP-command *abort*. Considering the current version of the system, the feature requires to check whether any file transfer is in progress or about to start, and if so, terminate the transfer and close the data connection.

Analysis of the new requirements with Theme/Doc resulted in one new theme, `abort`. Figure 4.9 shows a design for the theme. Again, we can see (the highlighted parts) that the design theme implicitly addresses two crosscutting concerns. Both

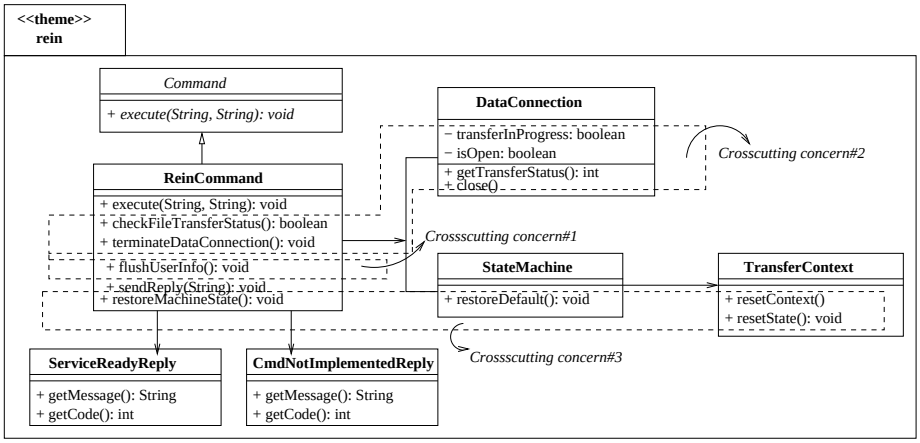

Figure 4.8: `Rein` theme implicitly addresses 3 crosscutting concerns.

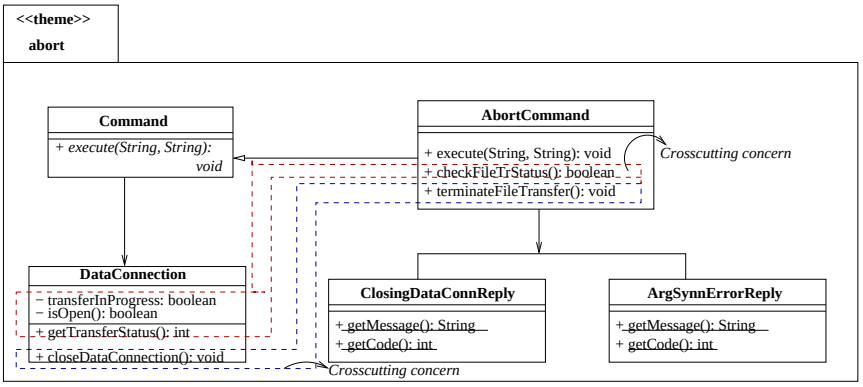

Figure 4.9: `Abort` theme implicitly addresses two crosscutting concerns.

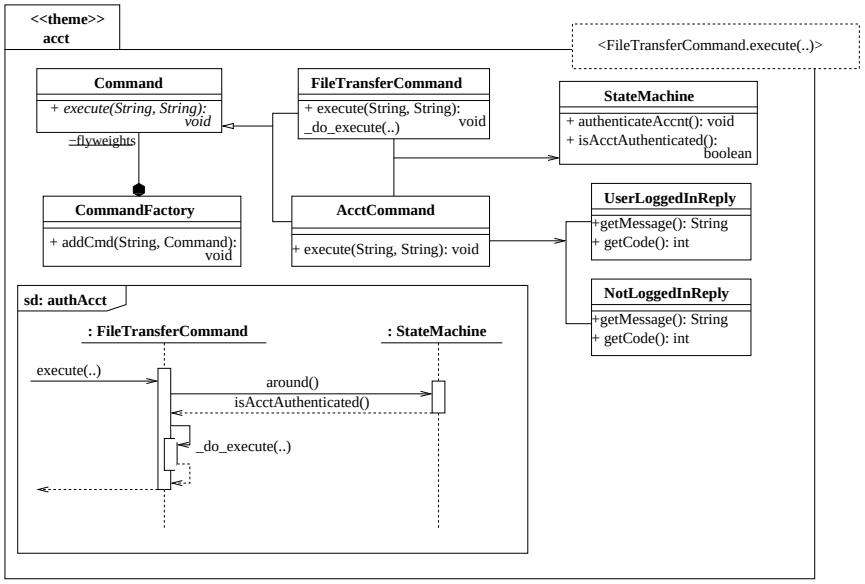

Figure 4.10: `Acct` theme implicitly addresses a crosscutting concern.

crosscut `rein` and `send_receive_files` themes, along with this newly added one (`abort` theme).

Had we processed all the requirements (new and existing ones) with Theme/Doc at this stage of development, we would have to invasively modify, as well restructure the existing separation of the base and crosscutting concerns. We could identify a new crosscutting concern relating to termination of a file transfer; separation of this crosscutting concern would require restructuring to different existing themes. The other crosscutting concern, relating to checking of an ongoing file transfer, was identified in the previous evolution step. To avoid the restructuring and invasive modifications according to the early aspect model of Theme, we had to implicitly address several crosscutting concerns and thereby, break the clean separation of the system.

4.3.4 Version 5

We attempted one more evolution step. For version 5 of the system, we considered inclusion of the feature corresponding to the FTP-command *acct* (account). The feature requires to authenticate a user with his/her corresponding account information, and verify the authentication before approving execution of certain FTP-commands (related to transfer of files).

Analysis of the new requirements with Theme/Doc resulted in one aspect theme, `acct`. Figure 4.10 shows a design for the theme. The theme encapsulated the corresponding crosscutting behaviour, and integration of the theme with the existing system resulted into a complete functioning system (Version 5). However, similar to the previous evolution steps if we consider the design of the current system as a whole, we would find a crosscutting concern being implicitly addressed by the theme (as shown in the figure); the behaviour for updating and verifying the machine state crosscuts the existing theme, `verify_authentication`, along with this new theme. Implicit consideration of the crosscutting concern within the new theme again, breaks the asymmetric modelling of the system.

4.4 Summary

We applied the early aspect model of Theme in developing the 'minimum implementation of FTP server' and then evolving it. In developing the original system (Version 1) we explored the consequences of development of individual themes, both independently and all at one time. We observed that considering development of

themes all at one time resulted in a successful integration into a complete functioning system. But independently developed themes, irrespective of pre-defining a design architecture, resulted in crosscutting differences. Without considering late treatment of crosscutting concerns, independent design themes did not suffice for a successful integration.

We attempted several evolution steps to the base system, considering it was developed along Path 1 (development of themes all at one time). In each evolution step, we encapsulated the new requirements into new design themes and integrated them with the existing system. However, considering the complete design of the system (after evolution) we observed that the newly added design themes had to implicitly address several crosscutting concerns, thereby breaking the clean separation of the system. Had we to preserve the asymmetric AOSD, we would have to invasively modify several parts of the existing system, as the new requirements crosscut several base and aspect themes; this would also cause restructuring of the existing base and crosscutting concerns. The first option violates a key principle of the early aspect approach itself (e.g., to maintain an asymmetric separation of base and crosscutting concerns) and as a result, cannot be considered feasible; the second option fails to achieve non-invasive evolution and hence, is avoided by the early aspect model of Theme. As a result, the early aspect model of Theme proved unsuccessful in supporting evolution of the system.

Our exploration and investigation of the early aspect model of Theme demonstrates that the early aspect model of Theme lacks support for the software properties of independent development and evolvability. In Chapter 6, we provide an analysis of the model based on each of the MC properties.

Chapter 5

Evaluation: Applying the late aspect model

This chapter describes application of the late aspect model of Theme in the development and evolution of an FTP server. Similar to our approach in applying the early aspect model, here we also explored and analyzed possible alternative decisions at each stage of the lifecycle. Section 5.1 discusses our approach in deriving a set of themes from the initial artifact of RFC 959min, Section 5.2 discusses the issues with development of the derived themes and their integration into a complete system, and Section 5.2.3 discusses the various evolution steps we attempted on the base system.

5.1 Decomposition into themes

The late aspect model of Theme defines a *theme* as [25, 22]: "a design unit that encapsulates a concern, which can be a system feature or a type of processing". The model suggests for decomposition of the system specification into individual features or tasks irrespective of the crosscutting behaviour [22]; each feature or task is expected to be mapped into a separate design theme. However, the Theme model does not prescribe any explicit mechanism to decompose the system specification into different system features (or tasks), to be encapsulated into separate themes. In this section, we discuss our derivation of individual themes from the system specification (RFC 959min) and justify the derived set of themes as a representative one for a decomposition of the system specification into a set of system features or tasks.

To derive individual themes from the specification document (RFC 959min), we analyzed the requirements in the document and extracted out possible system features. We considered each of the different FTP-commands supported by Version 1 of the system, as an individual system feature. We also considered listening to a port to connect a client, and processing requests from a client as separate features. In the process we derived 11 features, 9 of which involved individual FTP-commands (*user, port, type, mode, stru, retr, stor, noop* and *quit*) and the other 2 were for establishing connection with clients and for processing their requests. We mapped each feature onto an individual theme to be developed via Theme/UML.

We consider our set of themes as a reasonable one that represents a decomposition of the specification into a set of features or tasks. To justify this, we conducted a small survey with 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 the FTP specification document and was asked to decompose the system (Version 1) into individual tasks or features for separate sub-teams to work independently. From their responses, it was found that all of them suggested for distribution of different FTP commands among separate sub-teams, with a few of them also suggesting that the FTP 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 the clients and performing different client requests, varied in their suggestions. But all their responses aligned closely with our derived set of features. Based on the survey, we can assume that our set of themes represents a decomposition of the system into individual features or tasks. We proceeded with the development and integration of the derived (11) themes into a functioning base

system (Version 1).

5.2 Development and integration of the derived themes

Similar to our approach in applying the early aspect model, in developing the derived set of themes with the late aspect model we explored the consequences of two alternative paths: Path 1 considers development of all themes together, and Path 2 considers their independent development. In this section, we discuss feasibility of each path in the design and integration of the derived set of themes.

5.2.1 Path 1: Themes developed all at one time

We explored Path 1 considering designs of the derived themes all at one time. We present design details for a number of themes along this path, in Appendix B (Section B.3). The themes developed together conformed to a uniform communication protocol. Each design theme possessed the explicit knowledge about how to communicate with any other theme. Being developed all at one time, the themes did not result in any difference or conflict among their individual perspectives on the system. As a result, they could be successfully integrated into a complete system (Version 1) according to the Theme/UML composition mechanism. Path 1, applying the late aspect model of Theme, proved to be successful in integrating individual themes into a complete functioning system.

5.2.2 Path 2: Independent development of individual themes

In investigating independent development of individual themes according to Path 2, similar to our approach with the early aspect model, we explored the consequences of

two sub-paths: one (Path 2.1) considering independent themes not sharing a common design architecture, and the other one (Path 2.2) considering a pre-defined design architecture for individual themes to follow. In this section, we discuss the issues regarding the development and integration of individual themes along each of the sub-paths.

Path 2.1: Independent themes do not follow a defined design architecture To explore the consequences of independent development of themes, we considered possible alternative designs for each individual theme. We found that each theme could be designed in multiple alternative ways, each of which validly represented the encapsulating feature. In this section, we discuss some of the differences arising from independent designs of individual themes and show how the late aspect model addresses these differences to result in a successful integration.

Figure 5.1 shows two design themes `retr` and `port`. They encapsulate the corresponding features for the FTP-commands *retr* and *port* respectively. `Retr` theme captures the server behaviour to send a requested file and acknowledge with a corresponding reply-message. It expects to be communicated with by a call to `RetrieveCommand.execute(File, Session)` operation, with the requested file-name and the session value corresponding to a particular user-server connection, as arguments. The requested file is sent to the user via a data connection and a corresponding reply-message is sent via the control connection with the user. `Port` theme captures the behaviour of storing the port address of a remote-user and sending a corresponding reply message. The theme expects to be communicated with by a call to `PortExecuter.perform(String, String, Session)` operation, expecting

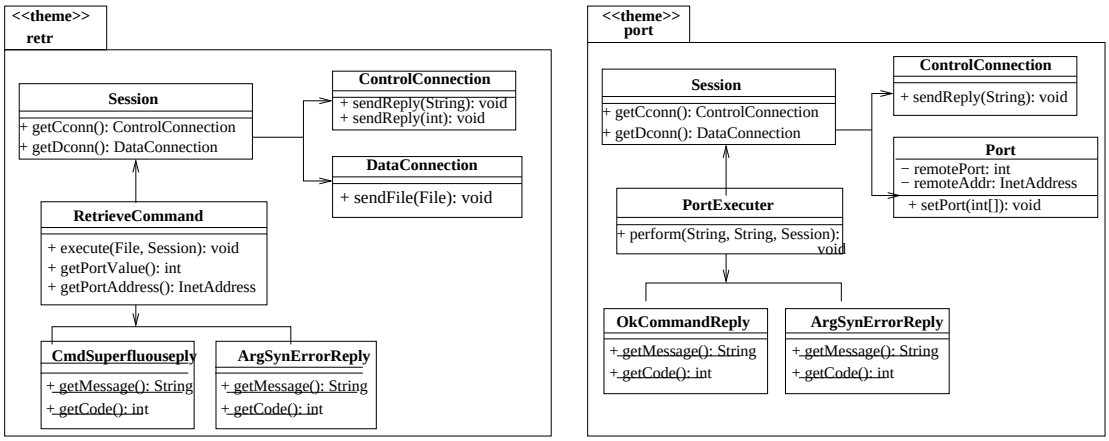

Figure 5.1: Design themes `retr` and `port` do not share a common design architecture

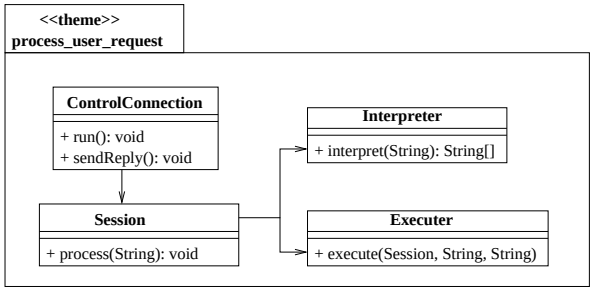

Figure 5.2: `Process_user_request` theme.

name of the FTP-command requested, the port information, and the session value as arguments. The remote user address and port information are stored into an instance (corresponding to a particular session with the user) of `Port` class. The design themes can be considered valid representations of the encapsulating features, since each theme captures the required behaviour as described in the corresponding feature description.

Similar to our experience in applying the early aspect model of Theme, we found that the independently developed themes along this path resulted in communication mismatches. Let us consider a design for `process_user_request` theme (shown in Figure 5.2) that captures the details of processing requests, sent by a user. The theme, upon receiving a user-request, interprets it to extract out the requested FTP-command and the associated argument value. Since the system behaviour corresponding to different FTP-commands are encapsulated into separate themes, the `process_user_request` theme need not address the details of an FTP-command. For any FTP-command, the theme invokes `Executer.execute(..)` operation; this invocation is expected to trigger an appropriate operation of a corresponding theme (that implements an FTP-command), during composition.

Let us consider a scenario when `process_user_request` theme receives a user-request for the FTP-command *retr*. Upon interpretation of the user-request, the theme would need to communicate with `retr` theme for execution of the FTP-command. But symmetric composition (with Theme/UML) of `process_user_request` (in Figure 5.2) and `retr` (in Figure 5.1) themes do not suffice for a meaningful communication between them, as the `process_user_request` theme does not possess the explicit knowledge about how to communicate with `retr` theme. `Retr` theme in Fig-

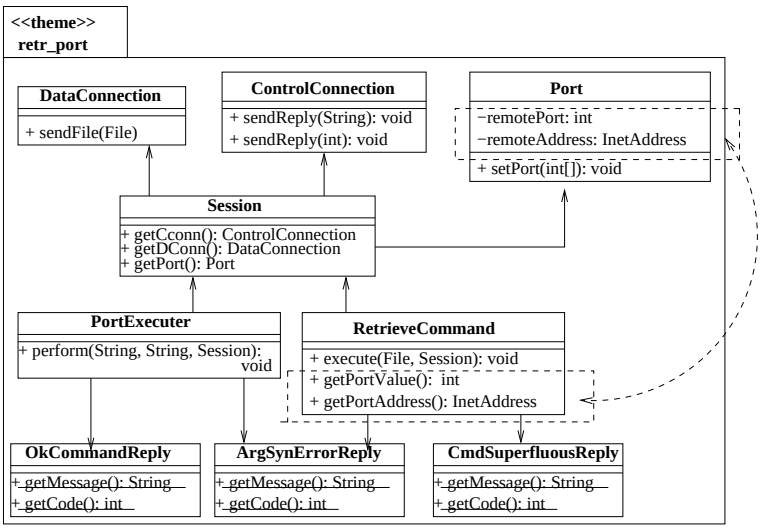

Figure 5.3: Composition of `retr` and `port` themes.

ure 5.2 expects to be communicated with by a call to `RetrieveCommand.execute(File, Session)` operation; this explicit knowledge, not known by `process_user_request` theme, the composition results in a communication mismatch.

Moreover, we observed that independently developed themes resulted in differences among their respective views. Consider `retr` and `port` themes in Figure 5.1. Figure 5.3 shows composition of the two themes according to the Theme/UML composition relationship of "merge", with "match-by-name" integration rule. The highlighted parts show the conflicts (differences) in views between the two themes. `Retr` theme requires the port information to send a file to a remote-user. But the theme (in Figure 5.1) does not have a view of the `Port` class; `port` theme stores the remote-user port information into an instance (specific to the session value) of `Port` class. To re-

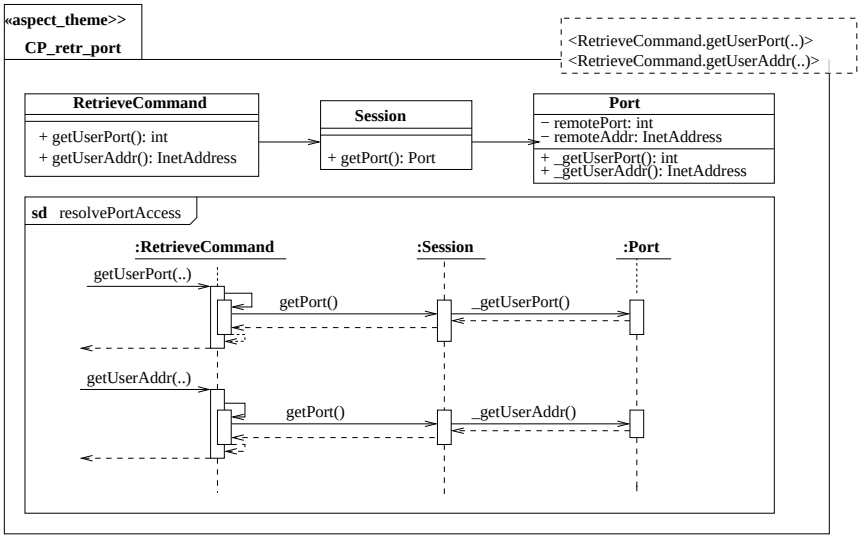

Figure 5.4: An aspect theme to address the crosscutting differences between `retr` and `port` themes.

sult in a meaningful composition, we would need to pass the correct port information (stored in an instance of `Port` class), whenever requested by the `RetrieveCommand` class. But the symmetric composition mechanism (of Theme/UML) would not suffice here, as we cannot match the `RetrieveCommand` class with the `Port` class, with any composition rule/relationship.

In our explorations of alternative designs along the path, we found the above mentioned two types of difficulties in integrating independent themes; one is the differences in views or conflicts between any two design themes and the other one is the communication mismatches among individual themes. Both the differences are crosscutting in nature that need to be addressed to form a functioning system. The

late aspect model of Theme, similar to any late aspect approach, suggests that the crosscutting differences should be addressed during integration of individual design models.

We attempted to address the crosscutting differences between any two composing themes by introducing "aspect themes" (similar to the ones in the early aspect model), during integration. Figure 5.4 shows an aspect theme to resolve the crosscutting differences between `retr` and `port` themes (shown in Figure 5.1). The sequence diagram shows how the references to `RetrieveCommand.getUserPort()` and `RetrieveCommand.getUserAddress()` operations return the corresponding operations of an appropriate instance of `Port` class. We explored alternative designs for each individual theme and found that the differences among any two themes (resulting from their independent designs) could be resolved during their composition by encapsulating the crosscutting differences into aspect themes.

To resolve the communication differences among individual themes, we attempted to capture the communication protocol as one crosscutting concern (encapsulated into an aspect theme). We could include individual aspect themes to capture communications between two themes during their composition. But we needed to establish a uniform communication protocol in order to form a complete functioning system, and addressing the communication with individual aspect themes proved to be complex and more work; interactions among the individual aspect themes also needed to be addressed to ensure a uniform communication protocol. We rather encapsulated the communication protocol into one aspect theme, `composer` (shown in Figure 5.5), during integration of all themes. The aspect theme in Figure 5.5, possesses partial knowledge about different individual themes and implements the crosscutting

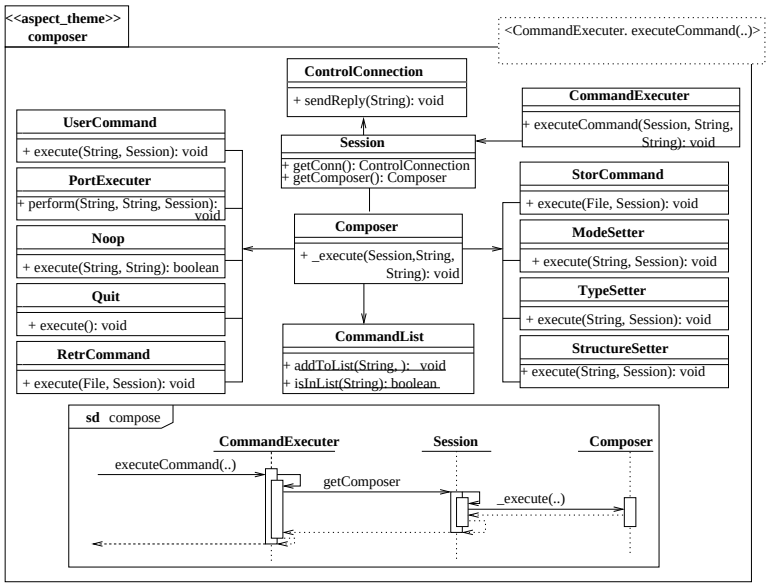

Figure 5.5: The aspect theme addresses crosscutting communications among individual themes.

communication behaviour. It intercepts a request to execute a particular FTP-command by `process_user_request` theme (execution of `Executer.execute(..)` method) and triggers the appropriate operation of a theme that encapsulates an FTP-command. With the knowledge about how to communicate with other themes, this theme could successfully resolve the communication differences in the integrated system.

We could resolve communication differences among individual design themes by adding an aspect theme. The conflicts and differences in views among independently developed themes could also be addressed by individual aspect themes, which encapsulated the crosscutting differences between any two themes during their composition. As a result, integration of all themes developed along this path, resulted into a complete functioning Version 1 of the system.

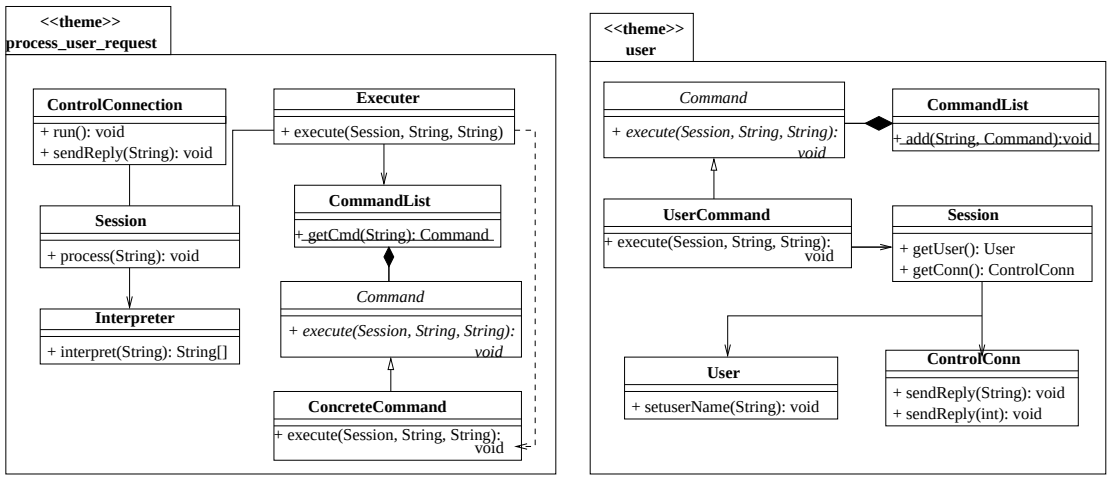

Figure 5.6: `Process_user_request` and `user` themes share a common design architecture.

Path 2.2: Independent themes follow a common design architecture

Expecting that a pre-defined design architecture can address the communication among individual themes, we explored the sub-path (Path 2.2) considering a common design for independent design themes. Similar to our approach in exploring the path applying the early aspect model, we decided that each theme encapsulating an FTP-command, would be designed following the Command design pattern [36]. The theme would also follow the Flyweight design pattern [36] to maintain the list of supported FTP-commands and for referencing an instance of a (corresponding) concrete `Command` class.

Figure 5.6 shows two themes that conform to the design architecture. The pre-defined architecture could address communications between the themes and compo-

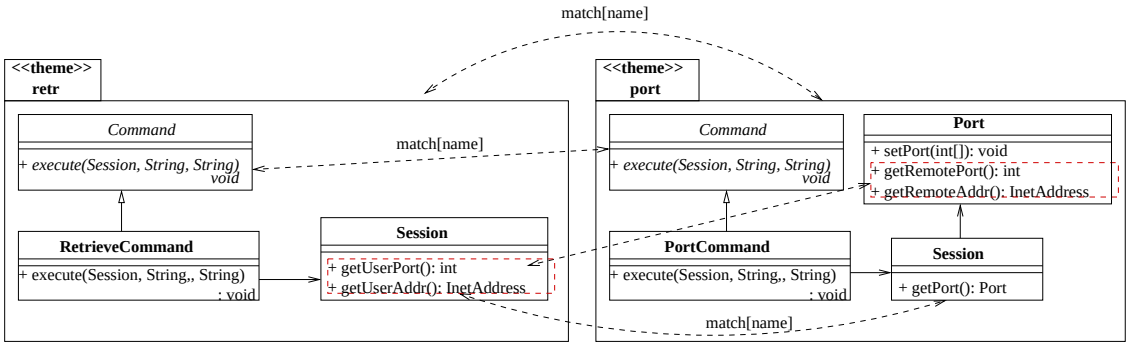

Figure 5.7: Partial views of `retr` and `port` themes developed along Path 2.2.

sition of the themes with the Theme/UML composition rules of "match-by-name" with "merge" relationship, sufficed for a meaningful integration. Details of integration of the two themes (in Figure 5.6) and their communication are discussed in Appendix B (Section B.4).

The pre-defined design architecture sufficed for resolution of the communication differences among individual themes. This could avoid the need for the inclusion of an aspect theme during integration to encapsulate the crosscutting communication protocol, as the common design architecture could resolve that up-front. However, in our explorations of alternative designs for individual themes, we found that consideration of independent development of themes, where individual themes conform to a common design architecture, still resulted in crosscutting differences and conflicts among the views of individual themes.

Figure 5.7 shows a partial view of two themes, `retr` and `port`, developed following the common design architecture discussed above. The highlighted parts (in red)

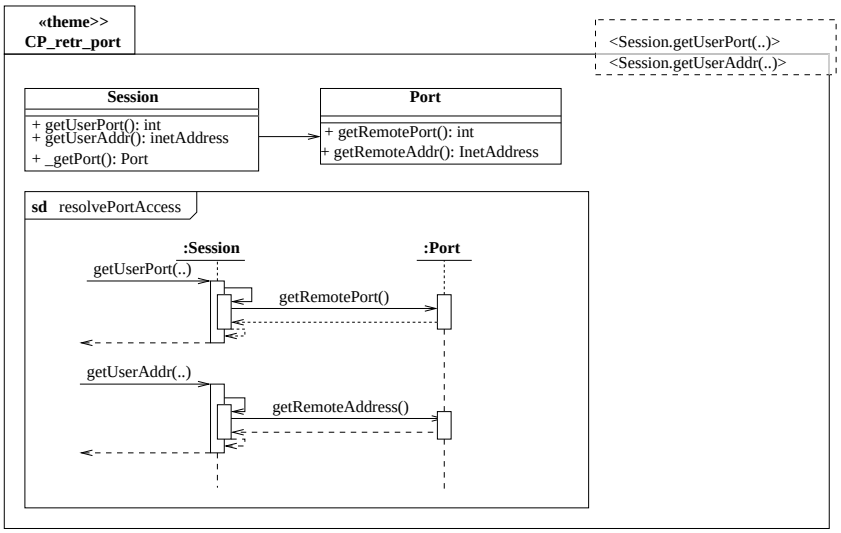

Figure 5.8: An aspect theme to address the crosscutting differences.

indicate the differences in their views that would result in an unsuccessful integration with Theme/UML. The `retr` theme need not know the details about how remote-user information (port value and address) is stored in the system; it only requires to access the values to send a file to the remote-user; the values are set by the `port` theme. Symmetric composition of the two themes in the figure (with Theme/UML) would not suffice for a meaningful integration, since we cannot match `Session` and `Port` classes with any composition rule or relationship. Such differences in views are essentially crosscutting concerns that need to be reconciled to provide a meaningful integration.

Following the late aspect approach principle, we addressed the crosscutting differences during integration of the themes. Figure 5.8 shows an aspect theme that

addresses the crosscutting differences to provide a meaningful composition of `retr` and `port` themes. The sequence diagram represents how a request for remote-user information triggers the corresponding methods in `Port` class.

In the same manner, we could address the crosscutting differences arising from independently developed design themes. The pre-defined design architecture could resolve the communication differences up-front; addressing of crosscutting concerns late during integration could resolve the conflicts among independently developed themes. Integration of the themes, developed along this sub-path, resulted into a complete functioning system.

To summarize the results of Path 2 with the late aspect model, we could successfully integrate the independent themes, developed with or without considering a pre-planned design architecture (either through Path 2.1 or 2.2). Along both the sub-paths, we were required to add aspects (during integration) in order to address crosscutting differences among the views of independent themes. In Path 2.1, we had to address communication differences among independent themes through encapsulating the communication protocol into an aspect theme. Path 2.2 could resolve the communication differences up-front, by considering a common design architecture for individual themes.

Both Path 1 and Path 2 can be considered successful, as we could integrate individual themes into a complete functioning base system. We attempted evolution of the system (Version 1), constructed along either of the paths, to investigate whether the late aspect model of Theme can address the evolvability property. In the next section, we discuss consequences of the multiple evolution steps that we investigated.

5.2.3 Evolution

We applied the similar evolution steps on the base system (Version 1), as we did in applying the early aspect model of Theme. This section discusses consequences of the different evolution steps.

For Version 2 of the system, we considered addition of the feature, "user-password authentication protocol". We evolved the system specification (RFC959min) by adding the new set of requirements. The added feature referred to support for *pass* command, new behaviour in the case of *user* command, and verification of authentication with user-password for different FTP-commands. We mapped the new feature into a new theme, "`user_pass_authentication`", and considered its development and integration with the existing system. As we discussed in the case of the early aspect model of Theme, in this evolution step we could map the changes to *user* command into the existing theme, `user`, or we could also consider it as part of the new feature, to be encapsulated into the new theme. Either decision did not have any effect on future evolution of the system.

Figure 5.9 shows a design for `user_pass_authentication` theme, which encapsulates all the details for the new feature (the theme also captures the modified behaviour for *user* command). `UserCommand` and `PassCommand` classes implement bahaviour for the FTP-commands *user* and *pass* respectively. `StateMachine` class captures the behaviour for verifying state-authentication for a user. `VerifyAuthentication(..)` operation returns either true or false based on the authentication state and the requested command name; certain FTP-commands require state authentication, others do not. For any unauthorized FTP-command request, it acknowledges the user with a corresponding reply-message.

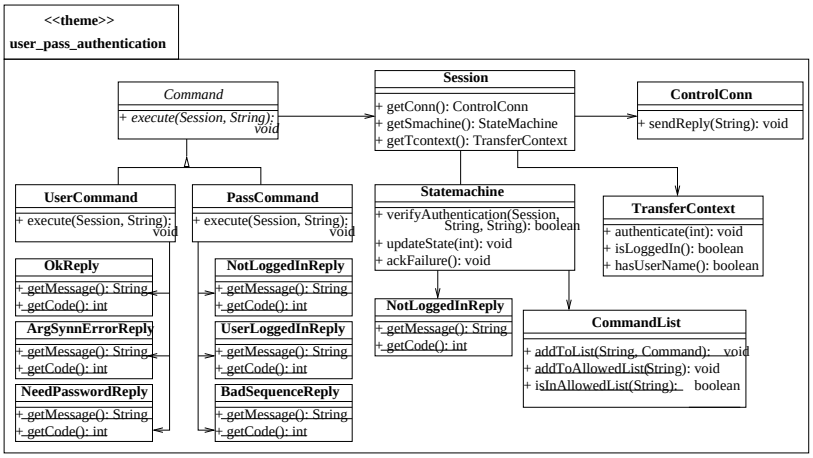

Figure 5.9: `User_pass_authentication` theme encapsulates the feature representing user-password authentication protocol.

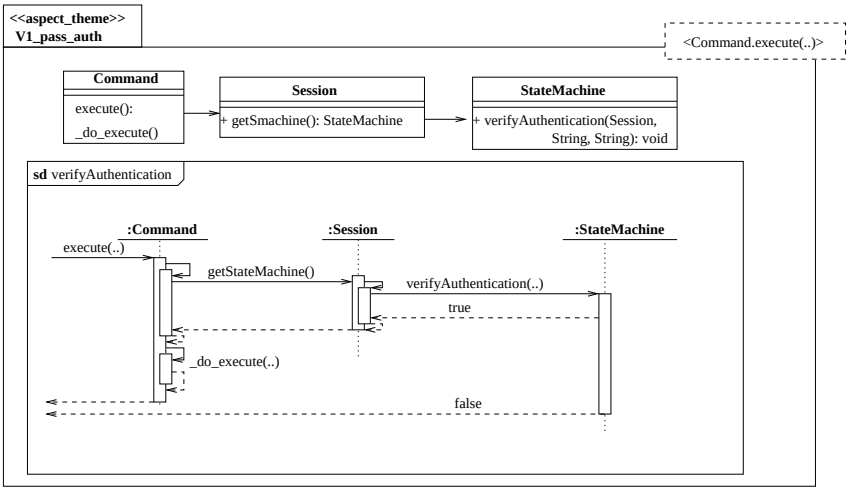

Figure 5.10: An aspect theme to address communication with the new theme.

We considered integration of the theme with the base system, developed considering a pre-defined design architecture (along Path 1 or Path 2.2). We observed that although the base system followed a common design architecture, communication between the new theme and the existing system could not be resolved with ordinary (symmetric) Theme/UML composition. This is because the original design of the system did not anticipate this evolution step. To address the crosscutting communication in integrating the new theme, we added an aspect theme, shown in Figure 5.10. The aspect intercepts execution of an FTP-command (the template parameter binds a method, `Command.execute(..)`), triggers the verification operation (`StateMachine.verifyAuthentication(..)`), and based on the verification result, allows or disallows execution of the FTP-command. The integration, considering addressing of the crosscutting concern, sufficed to result in a complete functioning system (Version 2).

We also investigated integration of the new theme with the base system, constructed considering independent themes not sharing a common design architecture (through Path 2.1). We explored alternative designs for the new theme and attempted integration with the existing system. We could successfully integrate the themes into a complete system (Version 2), considering the communication between the themes and the differences in their views are addressed via late aspects. The first evolution step proved successful, without requiring any invasive modification to the existing system.

For the next two evolution steps, similar to the application of the early aspect model, we analyzed inclusion of the features corresponding to the FTP-commands *rein* and *abort*. Similar to our explorations in designing a theme to encapsulate an

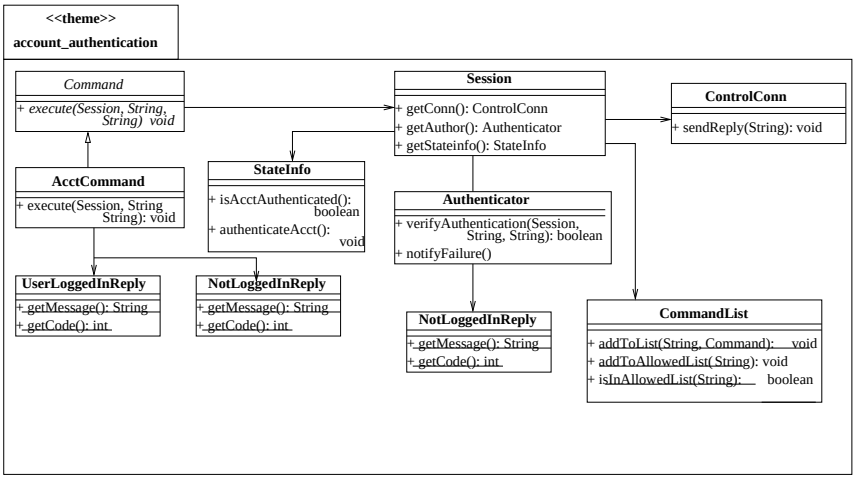

Figure 5.11: `Acct` theme addresses account-authentication feature.

FTP-command, we designed the two themes and attempted integration with the existing system. Following our late aspect approach of addressing the crosscutting differences with aspect themes, we could successfully integrate both the themes to construct Version 3 and Version 4, respectively. Appendix B (Section B.5) shows a design for `rein` theme and discusses its integration with the existing system, considering the late aspect approach. The evolution steps could be accomplished in a non-invasive manner, by adding new themes only.

For Version 5 of the system, we included the feature to support "user-account authentication protocol", corresponding to the FTP-command *acct*. We considered encapsulation of the feature into a new theme, `acct`, and designed the new theme with Theme/UML. Figure 5.11 shows a design for the theme. `AcctCommand` class implements behaviour for the FTP-command, *acct*; for a valid account log-in attempt, the account-authentication information (e.g., update of the state) is stored in

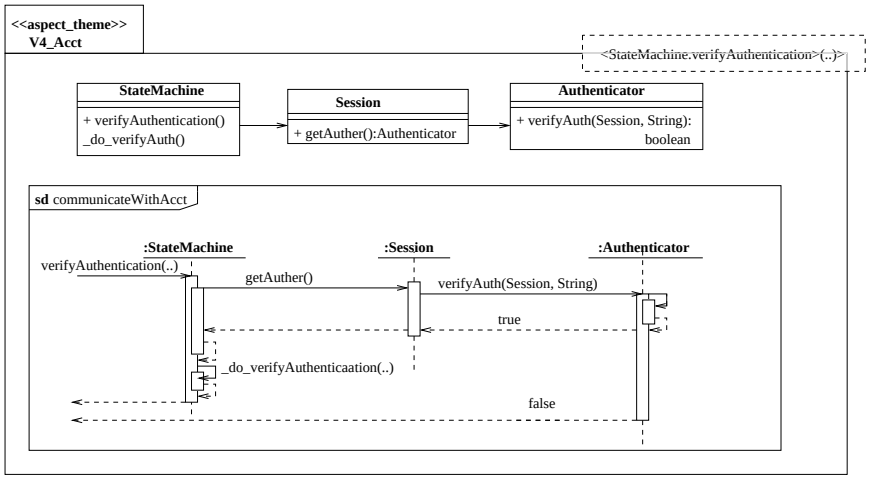

Figure 5.12: An aspect theme to address the communication with `acct` theme.

`StateInfo` class and the user is acknowledged with a reply message. `Authenticator` class implements the behaviour for verifying an FTP-command name with the account state. The FTP-commands relevant to file transfers would need authentication; other commands would be allowed to be executed without account-authentication.

In integrating the new theme with the existing system, we realized the need for defining a communication protocol for the interactions between the (existing) password-authentication operation and the behaviour for verifying account-authentication. The communication among them should occur in such a way that for an FTP-command, prior to verifying password-authentication, the system would verify account-authentication. This communication protocol could not be defined within the new theme, since the new theme was supposed to encapsulate the details corresponding to the new feature only, without worrying about details of other design themes. We followed our late aspect approach of including an aspect theme (to address the

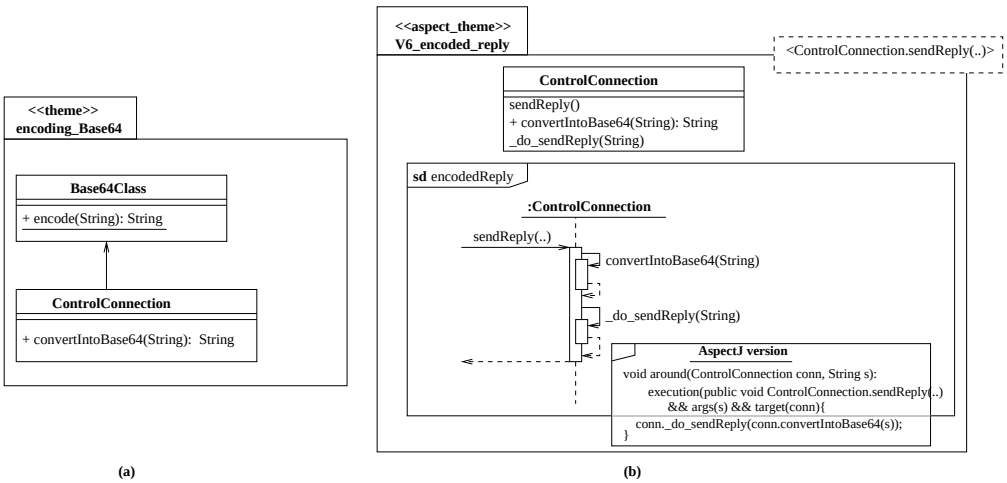

Figure 5.13: `encoding_Base64 theme` encapsulates the added behaviour

communication) in integrating the new theme with the existing system. Figure 5.12 shows the aspect theme. The template parameter binds a password verification operation and triggers the account verification operation. In the case of an (account) unauthenticated file-transfer request, the triggering operation is not executed and instead returns to the base flow of operation; otherwise the flow of control proceeds with the password verification operation.

The late aspect approach, with inclusion of an aspect theme, sufficed for integration of the themes into a complete Version 5 of the system. If we consider the complete system (Version 5), the newly added feature introduced a number of cross-cutting concerns into the existing system. However, addition of the new feature with the existing system did not require any invasive modification; rather it was possible in an additive manner, similar to the previous evolution steps. The late aspect model of Theme proved successful in achieving all four evolution steps (Version 2 to 5) in a non-invasive manner.

So far, the evolution steps examined how the late aspect model could handle addition of new system features. To investigate how it would perform in incorporating modifications to a crosscutting concern in the existing system, we explored one more evolution step. For version 6 of the system, we added a requirement stating that "*replies to the user should be encrypted in Base64 format as a security measure*". This requirement would have been mapped into `send_reply` aspect theme, in the case of the early aspect model; localized changes to the theme would have sufficed for a successful evolution in this situation. But we could not achieve such localized changes in the case of the late aspect model, as this crosscutting requirement had not been separated out as an individual theme in the existing system; it would therefore be interesting to explore how the model would fare incorporating the evolution step.

In order to continue with non-invasive evolution, we added an extra theme to encapsulate the changed behaviour. Figure 5.13(a) shows a simple theme that captures the behaviour for encrypting a message into Base64 format. The theme only addresses the details corresponding to the new feature, not worrying about its communications with the existing system. Next, we attempted integration of this theme with the existing system. According to the late aspect approach, we added an aspect theme to address the communication between the new theme and the existing system. The aspect theme in Figure 5.13(b) shows the communication protocol; a request to send a reply-message triggers the encryption operation. The "sendReply(..)" operation then continues with the encrypted message. Thus considering the late aspect approach, we could construct Version 6 of the system to incorporate modification to an existing crosscutting behaviour without any invasive modification to the existing system. Such additive evolution also provides rich support for the (un)pluggability

property of the system behaviour.

The evolution steps have demonstrated that the late aspect model of Theme can provide support for non-invasive evolution of the system in adding new requirements, as well as modifying existing crosscutting concerns, supporting the evolvability property.

5.3 Summary

In applying the late aspect model of Theme in the development and evolution of an FTP server, we investigated the consequences of the development of individual themes both, independently and all at one time. We could integrate individual themes into a complete functioning base system, considering either path. Independently developed themes resulted in crosscutting differences among their individual views of the system and also among their communications; development of themes all at one time could avoid such differences to result in a complete system with the Theme/UML composition mechanism. The differences among independent themes could be addressed considering late aspects. We could address the communication differences, as well as the differences among the views of independent themes (arising from the low level design details), introducing aspect themes during integration. We also observed that a pre-defined design architecture could resolve the communication differences up-front, relieving the need for addressing the communication protocol with late aspects.

Our investigation of the evolvability property of the model through multiple evolution steps demonstrates that new features, as well as modifications to existing

crosscutting concerns, could be encapsulated into new themes and integrated with the system considering late aspects. Regardless of whether the original system was constructed considering a common design architecture (to resolve communication differences up-front) or not, we were required to address the crosscutting communications between a newly added theme and the existing system during an evolution step. All the evolution steps could be accomplished by considering late aspects, to address the crosscutting communication among individual themes, as well to resolve the differences among their respective views. The non-invasive evolution steps indicate support for evolvability on part of the late aspect model of Theme. The additive evolution steps also indicate support for (un)pluggability of a system feature. In the next chapter, we summarize the findings and analyze the results with respect to the MC properties.

Chapter 6

Analysis and Discussion

The case study was performed with a view to evaluate the early and the late aspect models of Theme with respect to the MC properties. In the study, we performed development and evolution of a benchmark system with both the models and investigated consequences at each of the lifecycle stages. In this chapter, we provide an analysis of the results considering the software properties of interest and discuss how the study can represent a general evaluation of early and late aspect approaches. We also discuss some key details of the study, as well as its limitations and the potential future work.

6.1 Analysis of the models

In applying the early and the late aspect models of Theme, we investigated the MC properties of traceability, comprehensibility, independent development, and evolvability, based on the criteria described in Chapter 4. In this section, based on the results of the case study, we provide a comparative analysis of the two models of Theme, with respect to each of the MC properties.

6.1.1 Traceability

The early aspect model of Theme processes the system requirements early in the lifecycle and extracts out the crosscutting requirements from the base ones. The

processed system specification hence contains a set of base and crosscutting requirements, structured separately. Each requirement (base or crosscutting) in the processed specification, is mapped onto one design theme. As a result, a requirement in the specification document can be traced directly into the design model and the corresponding implementation artifact. Similarly, each design or implementation model could be traced back to the encapsulating requirement(s). Therefore, the property of traceability is successfully addressed by the early aspect model of Theme. However, in order to maintain traceability of individual requirements across the lifecycle artifacts, asymmetric separation of the base and crosscutting concerns has to be maintained throughout the lifecycle. We observed in the study that preservation of asymmetric modelling (throughout the lifecycle) on the part of an early aspect approach, comes at the cost of evolvability; as a result, traceability of crosscutting requirements may not be maintained over the evolution of the system.

The late aspect model of Theme does not prescribe any requirements engineering approach to process the system requirements, or to separate out the crosscutting ones. Rather, individual system concerns are identified from the requirements' description. Each individual concern is mapped onto a separate design theme and its corresponding implementation. As a result, each requirement corresponding to a system concern should be traced across the lifecycle artifacts. However, it should be noted that the model provides traceability for any requirement that is modularized in the specification document (or in the system's description). A requirement, which remains crosscutting in the specification document, may not be traced directly in the design or the implementation.

6.1.2 Comprehensibility

The results regarding comprehensibility were mixed in the case of both the Theme models. In this section, we first discuss the development stages where improved comprehensibility can be achieved with either of the models and also the stages where the property remains doubtful. Next, we discuss the practices that may help providing better support for the software property.

According to Parnas's definition of comprehensibility [64] (e.g., comprehensibility ensues from modularity of concerns), both the models fare equally in addressing the software property. With individual requirements (in the specification document) being mapped directly into a design theme, it can be expected that to deal with a particular requirement, one should only need to look into the corresponding theme (design or implementation). This should lead to better comprehensibility for the maintenance team, since one need not understand the complete system design (as in the case of an OO design model) to deal with a particular system requirement. Development of an individual concern on part of a development team should also be simpler since, the team only needs to consider the details corresponding to that concern, without worrying about the remainder of the system. This applies to both the models of Theme.

However, comprehensibility also refers to the complexity or the ease of development. In our study, we found that development of an aspect theme in the early aspect approach might be more complex, since it requires explicit knowledge about the base, as well as other aspect themes. In the case of the late aspect model of Theme, we experienced that composition of individual themes can get complicated, when consid-

ering identification of the crosscutting differences to reconcile (with aspect themes), during integration. Identification of the differences and conflicts among individual themes may require understanding of the fine-grained details of their designs.

In the study, interaction diagrams for individual themes proved useful in identifying the conflicts and differences during integration. Usually conflicts occur from the discrepancies between views across a theme's boundary; the differences in expectations on the part of a theme from the remainder of the system are non-trivial crosscutting concerns. With the interaction diagrams, we could identify the differences among the views of different themes, during their composition. However, our study involved a small system; for large, complex systems, interaction diagrams might not suffice for identification of all possible conflicts among a large number of (independently developed) themes. There might be the need for an explicit specification mechanism, by which individual themes may express what they expect from the remainder of the system, and where possible conflicts may arise.

6.1.3 Independent development

To simulate results for independent development of individual themes with either of the Theme models, we considered multiple alternative designs for each design theme that correctly represented the encapsulating requirements. The results indicate that the early aspect model of Theme lacks support for independent development of individual themes. Regardless of providing a pre-defined design architecture for individual themes to follow, or avoiding the up-front effort in such pre-planning, we found that individual themes resulted in differences among their views; the differences resulted from low level design details of individual themes that could not be

foreseen prior to their designs. The differences need to be resolved to result in a complete system, but the early aspect approach does not prescribe any means to address the crosscutting differences resulting after the designs. As a result, the early aspect model of Theme proved insufficient to support independent development.

The late aspect model of Theme, by addressing crosscutting concerns during composition of individual themes, proved successful in supporting integration of independently developed themes into a complete functioning system. We considered two alternative paths in investigating independent development of themes. When considering that individual themes do not follow a pre-defined design architecture, we observed that individual design themes resulted in differences among their communications with one another, as well as conflicts among their individual views on the system. These differences are essentially crosscutting concerns and according to the late aspect approach, are to be addressed during integration. We introduced aspect themes to address the crosscutting differences, and integration of the themes resulted in a complete system.

We also explored a path considering a pre-defined design architecture for individual themes. We observed that communication differences among individual themes could be resolved up-front by providing a common design architecture. As a result, we did not need to address the crosscutting communication protocol during integration; addressing the conflicts among the views of individual design themes (with aspect themes) sufficed for a successful integration into a complete system. However, the results indicate that the path following a pre-defined design architecture would lead towards a less feasible solution, compared to the other path. A specific design architecture can prove problematic for future evolution, as a change to the architec-

ture would require widespread modifications to multiple themes that conformed to it; defining a common design architecture has often been criticized as a less desirable software practice [92]. Moreover, in the study we found that even after developing the base system considering a pre-defined design architecture for individual themes, we had to address the crosscutting communication protocol to integrate a new theme during an evolution step; the design architecture, defined considering the original version of the system, did not suffice to address communication for future changes. As a result, avoidance of the up-front effort in pre-defining a design architecture and leaving the communications among individual themes to be addressed late during the integration, should lead towards a more feasible solution.

6.1.4 Evolvability

To investigate the property of evolvability, we attempted several evolution steps with each of the Theme models. The results demonstrate that the early aspect model of Theme fails to address the software property, while the late aspect model proved feasible to address evolution of software systems.

In applying the early aspect model of Theme, we experienced that an evolution step may introduce concerns that crosscut a number of existing base and aspect themes. In order to maintain an asymmetric separation between the base and crosscutting concerns, we would have to restructure the existing separation and invasively modify different parts of the system. Since this indicates poor evolvability, the early aspect model of Theme avoids consideration of the existing requirements, and suggest for encapsulating the new requirements into new design themes; integration of the new themes with the existing system is expected to accomplish evolution in a

non-invasive manner. However, we observed that in following the approach, we had to consider several crosscutting concerns from within new design themes. Implicit consideration of crosscutting concerns breaks the existing asymmetric separation; this is a contradiction to the early aspect approach itself, since the key principle is to maintain an asymmetric separation between the base and crosscutting concerns throughout the software lifecycle. Thus, the early aspect model of theme proved to fail addressing evolvability.

The late aspect model of Theme on the other hand, proved to provide support for non-invasive evolution for the system. We could include new features (that had not been anticipated previously) into the system in an additive manner; changes to the crosscutting behaviour in the existing system could also be incorporated without any invasive modification. Non-invasive evolution of the system indicated support for evolvability on the part of the late aspect model. The additive evolution steps also indicate the model's support for (un)pluggability and reusability of individual system concerns.

Since future changes may not be anticipated in developing a software system, early separation of crosscutting concerns can lead to poor evolution. Avoidance of separation of crosscutting concerns until development of individual design models, and only addressing crosscutting concerns during the integration phase, can provide better support for the software property of evolvability.

6.2 Discussion

In this section, we discuss how an evaluation of the Theme models can represent evaluation of early and late aspect approaches in general. We also discuss some key points of the study, its limitations, and directions for future work.

The early aspect model of Theme follows a requirements engineering approach, which, similar to a general AORE approach, identifies and separates out crosscutting concerns from the base system. The results of the study indicate that non-invasive evolution may not be achievable by any early aspect approach, without breaking the asymmetric separation of the system. An evolution step may introduce new crosscutting concerns that were not existent in the existing system. To maintain the asymmetric modelling, an early aspect approach would have to consider modification and restructuring of the base and crosscutting concerns, adversely affecting evolvability. With a view to addressing evolvability, the early aspect model of Theme attempts to incorporate evolution in an additive manner, by encapsulating new requirements into new design models that should be able to avoid any invasive modification to the existing system. However, the results demonstrate that the model fails to provide evolvability, while maintaining an asymmetric separation of the base and crosscutting concerns, which is an essential criteria for an early aspect approach. The results should hold for any early aspect approach in general; in addressing evolvability, any other early aspect approach should fare equally or worse compared to the Theme model.

The late aspect model of Theme is representative of a symmetric decomposition approach, which incorporates the late aspect concept of addressing crosscutting

concerns during the integration phase. Results for the MC properties in evaluating the late aspect model of Theme should generalize to those for other symmetric AOSD approaches, which consider crosscutting concerns late in the lifecycle. However, evaluation of the late aspect model of Theme for all the MC properties, may not generalize for the late aspect approaches that do not follow a symmetric AOSD.

Evaluation of the two models of Theme can therefore be considered as an initial evaluation of early and late aspects (that follow symmetric AOSD) in general.

We realized several limitations of the study, in evaluating the Theme models. The development process being conducted by a single developer, investigation of the property of independent development was problematic. To overcome this limitation, we considered multiple alternative design decisions for each theme; the possible alternative design decisions were presumed to represent (or simulate) the differences in designs by separate sub-teams. However, this might not be representative of an independent or distributed development environment, in general.

Another key issue was the small-sized system. We considered FTP server as a standard benchmark system to investigate the MC properties; whether the results regarding the MC properties would be similar in case of large and complex systems, might be questioned. However, the results do indicate that problems with the early aspect approach (addressing evolvability and independent development) should be worse for a large, complex system; the results for the late aspect approach indicate that consideration of crosscutting concerns should provide support for evolvability and independent development (along with traceability and comprehensibility), even for large systems. Conversely, identification of crosscutting differences during integration of large systems (especially in distributed development environment) might

prove a lot of work, requiring finer-grained details about individual system parts.

The late aspect approach proved to be more feasible in addressing software evolution. But the study may not provide a generalized evaluation for software systems of all domains. The late aspect approach might be well suited to support systems that are deployed in rapidly changing environments. For fixed-scope, as well as mission-critical systems, the late aspect approach might not prove effective, since support for unanticipated changes may not be the key goal for those systems.

In our evaluation, we did not investigate the performance issues. Integration of individual design themes might result in degraded runtime efficiency, for example. For both the approaches, efficiency of the tool support in integrating individual design models would be important as well. Whether the approaches can support development and evolution of performance critical systems, would be interesting to investigate.

AspectJ proved useful in addressing crosscutting concerns both early and late in the lifecycle. However, in providing the symmetric composition of any two design models according to the Theme/UML mechanism, the use of AspectJ proved more work, as it was not originally intended for the symmetric composition. Use of a tool like Hyper/J for symmetric composition, and use of AspectJ for asymmetric composition, might lead to a more feasible solution.

In our evaluation, the late aspect model of Theme proved to be better and more feasible in addressing the MC properties. However, considering development of themes all at one time, the early aspect model might prove to be less work (e.g., more feasible), than the late aspect one. In the early aspect model, crosscutting concerns are separated out up-front. As a result, individual base themes need not

consider details of the crosscutting behaviour. But in the late aspect model, each design theme considers crosscutting concerns implicitly. This might lead to overlapping of effort, as well as redundancy of work. In the study, we ran into the issue of determining how many details a theme should possess about a shared concern (crosscutting behaviour). For example, it proved redundant for each design theme (in the late aspect model) to implement behaviour for sending a reply-message to the user. But it was not apparent whether one theme should encapsulate the behaviour and other themes should just refer to it without implementing the details, or should each theme implement the behaviour considering that the conflicts would be reconciled during integration.

6.3 Summary

Through applications of the early and the late aspect models of Theme, we have provided an initial evaluation of when crosscutting concerns should be addressed in the software lifecycle in order to provide the MC properties. The initial evaluation demonstrates that crosscutting concerns should be addressed late in the lifecycle; early separation of crosscutting concerns according to the early aspect approach, would lead to failure. The negative results with the early aspect model of Theme should hold for any early aspect approach in general. The positive results with the late aspect model of Theme should generalize to a late aspect approach in general, which considers symmetric modelling of concerns; but the assessments cannot be generalized to the late aspect approaches that do not consider symmetric AOSD. A comparative analysis of late aspect approaches that consider symmetric modelling of

concerns, with the ones that do not consider symmetric AOSD, would be interesting; the results would help determining whether symmetric AOSD should be adopted or avoided, in order to better address the MC properties on the part of a late aspect approach.

Chapter 7

Related Work

Different AOSD approaches have been promoted based on either of early or late aspect approaches. But no work to date has addressed the question of whether either approach can provide better support for software evolution. In this chapter, we discuss the related work, categorizing them into three sections: work on early aspects, work on late aspects, and empirical evaluations of different AOSD approaches.

7.1 Early Aspect Approaches

AOSD work on the early aspect approach has focused on means to explicitly separate out crosscutting concerns from the base system, early in the requirements engineering phase. The process to identify and separate out crosscutting concerns from the system requirements is commonly termed as, "aspect-oriented requirements engineering". The different early aspect approaches and work groups can broadly be categorized as: component-based AORE, goal-oriented AOSD, viewpoint-based aspects, use cases and scenario-driven AOSD, architecture-oriented aspects, and aspectual formal specifications, etc. We discuss each of the categories in the remainder of the section.

Work by Grundy [37, 38] on AORE for component-based systems, is one of the initial work that motivated early aspects. The technique is based on separation of crosscutting concerns, especially the non-functional requirements from the system

components. The process analyzes requirements to identify candidate components, and each component is then analyzed to identify the aspects. Aspects for components and component groups are refined and aggregated, and the derived components and aspects are verified against the systems requirements. The approach, however limits itself to the requirements engineering phase only, not providing any direction for design and integration of the components or the aspects.

The viewpoint-based AOSD approach [72, 6] has been promoted via Arcade [70], which extends the traditional viewpoint based approach of the PREview model [79]. Arcade provides separation of aspectual and non-aspectual requirements as well their composition. It uses XML to represent the artifacts of concerns, views, and requirements that are provided as inputs to the PROBE framework [47, 48]. PROBE establishes mapping of aspectual requirements throughout the next stages of the lifecycle, providing traceability of requirements from the requirements documents to the design and implementation. Later work [75] on the approach has focused on improvement of the requirements engineering process for better identification and separation of the aspectual requirements from the non-aspectual ones. The viewpoint-based approach provides a platform for separation of the crosscutting concerns from the requirements. It aims at providing traceability, as well as composability of the aspects and the non aspects. But the approach does not address issues regarding evolution. It is unclear whether one would need to start from scratch or proceed differently, to incorporate new requirements or to modify the existing ones, either of which may alter the structures of the aspects and the non-aspects.

There has been initial work [99] on goal-oriented requirements analysis considering the early aspects. The approach uses a number of processes to identify and

separate out aspects from the functional and non-functional (soft) system goals. The *decompose* process decomposes the initial set of system goals and softgoals into subgoals, sub-softgoals, and their operationalizations, the *correlate* process relates the initial goals with the decompositions, the *resolve conflicts* process reconciles conflicts among goals/subgoals, and finally the *aspect finder* process identifies and separates out the aspects from the goals. The work is still in its initial stage, focussing on the requirements engineering phase only to identify the early aspects; the issues regarding the next stages of development are yet to be addressed.

Moreira *et al.* [60] propose use case driven requirements engineering approach to provide explicit separation of the quality attributes, as aspects. Functional requirements are represented using use cases, and the quality attributes are identified and described using special templates. A set of models represent integration of the crosscutting quality attributes with the functional requirements. Their later work [5] extends this approach to provide a more comprehensive identification and separation of aspects from the base requirements. The proposed model uses the common Use Case approach [45] to identify the system functionalities. These use cases are then refined to externalize the crosscutting functionalities with *include* and *extend* use cases. Non-functional requirements are identified and represented as templates. An extension of the use case model provides integration of the templates with the functional use cases. In the process, the model identifies *candidate aspect use cases* and determines the set of aspects from them. Similar work on early aspects have been promoted by Whittle and Araujo [96, 7] via scenario-based aspect modelling. The approach relies on use cases to capture both the functional and the non-functional requirements, and the crosscutting scenarios are identified from them. It uses a

state machine synthesis algorithm [97] to convert the non aspectual and the aspectual scenarios into respective finite state machines. A binding specification provides composition of the state machines. The approaches aim at providing comprehensive means to identify crosscutting requirements early in the lifecycle. Comprehensive modularization of aspects should be able to better address software evolution.

There has been some work on architectural aspects [58, 98] that focus on separate modelling of crosscutting architectural views from the non-crosscutting ones. A study by Xu *et al.* [98] extends C2 architecture [86] to demonstrate how the crosscutting concerns can be identified early in the lifecycle, and be modelled at the architecture level, separate from the base functionality of the system. The work in this field is still not matured and has so far considered only a number of non-functional requirements as probable aspects. Moreover, the work has not addressed whether (or how) such explicitly defined architectures can help independent development, provide reuse of components, or support unanticipated changes.

Mousavi *et al.* [61] have proposed an aspect-oriented formal specification framework for distributed real-time systems. The framework extends Gamma [9], a formalism based on multi-set rewriting on a shared data-space, intended to provide support for parallel/distributed architectures. The work focuses on separating out concerns of computation, coordination, distribution, timing, fault-tolerance, and persistence; the concerns are specified formally with a view to facilitating formal verification for specification-driven design and development. The work also provides a weaving mechanism to provide composition of functional and crosscutting concerns into a single semantic framework.

Some AOSD work has attempted to evaluate different AORE approaches. Chitchyan

et al. [20] provides a comparative analysis between traditional requirements engineering approaches and different AORE ones. The analysis mostly focuses on identification and separation of aspects from functional and non-functional requirements on part of the approaches, and comparison on how each approach fares with regard to the traceability of the requirements. The study also argues for or against the approaches regarding their respective scalability and evolvability. Bakker *et al.* [8] also provides a comparison among different AORE approaches evaluating properties like identification of aspects at the domain and the requirements analysis phases, modelling of system concerns, traceability, and tool support. The results of such evaluation attempts are argumentative since there has not been any empirical assessment or practical application in the evaluation process. Moreover, the analysis mainly focussing on how well an approach can identify the crosscutting concerns, issues regarding evolution of the system in the long run, has not been addressed.

7.2 Late Aspect Approaches

The late aspect approach has been motivated mostly through the work on symmetric AOSD. The symmetric approach is based on different decomposition techniques that aim at breaking the tyranny of the dominant decomposition (of OO) [34]. It is expected that symmetric design models would result in crosscutting differences; addressing of crosscutting concerns late in the lifecyle (via late aspects), is expected to provide better support for software evolution. The late aspect approach has also been promoted by different work that do not consider the symmetric AOSD, but address the crosscutting concerns late in the lifecycle. In this section, we first dis-

cuss different symmetric AOSD techniques that motivate the late aspect approach. Then we discuss other AOSD techniques that address crosscutting concerns late in the lifecycle, without considering the symmetric modelling of concerns.

Work by Shilling and Sweeney [76] on "views" can be considered as one of the initial work on symmetric modelling. The approach considers decomposition of a system based on conceptual slices or *views* of the user or the developer. A single object can participate in multiple view classes, which are defined as sets of ordered pairs of the form (*object class, interface*). Different views would typically overlap; the object class specifies how view instances interact by its rules for sharing and accessing instance variables. Composition of the objects is done by joining of each object instance to the view instance. The work however, does not elaborate on, or specify interactions among different views; how the crosscutting overlaps among individual views should be addressed during their composition, has not been specified either.

Contemporary work on Contracts [43, 44] has promoted the symmetric approach through interaction-oriented decomposition of the system. A system is considered to be decomposed into "contracts", where a contract defines a set of communicating participants and their contractual obligations. A contract also defines preconditions on participants required to establish the contract, and the invariant to be maintained by the participants. The specification of how a particular class implementation meets a participating object's obligation is declared via a *conformance declaration*. With the operations of *refinement* and *inclusion*, contracts provide means to create and reuse large grain abstractions based on behaviour. Refinement allows specialization of contractual obligations and invariant of constraints through extension of the

basic contracts, and inclusion allows contracts to be composed from simpler sub-contracts. Behavioural compositions of objects that participate collaboratively, are done through instantiation of the contracts.

Feature engineering [90, 89] also provides a symmetric decomposition of the system. The approach considers "features" as first-class constructs, where a feature is defined as [90]: "*a clustering of individual requirements that describes a cohesive, identifiable unit of functionality*". The approach is based on the perspective of identification and abstraction of features throughout the lifecycle, from requirements engineering to maintenance. Such modularization is presumed to provide better understanding and handling of architectural design, traceability, testing, reverse engineering, and configuration management. But the issues regarding interactions among features, and the composability or the resolution of non-composable features have not been addressed comprehensively in the work.

OORam [73], proposed by Reenskaug *et al.*, introduces role modelling through decomposition of a system into a number of distinct models. The models represent separate tasks or activities, supported by the different roles objects play in separate models. Integration of different roles into a complete system is performed via *synthesis* of the base models into a derived model. To ensure dynamic correctness of the derived model, *safe synthesis* through *activity superposition* and *activity aggregation* have been proposed. Superposition ensures that a base model activity is retained unchanged in the derived model, and aggregation allows changes to a base model activity to include interactions with other base model activities in the derived model. Catalysis [32] provides a UML based approach to role modelling. A system is considered to be composed of vertical and horizontal slices, where vertical slices represent

the views of different categories of users, and horizontal slices represent communication protocols and technical infrastructures. A system is thus decomposed along vertical and horizontal slices into separate role models. Integration of the role models are based on a UML *import* relationship, named *join*. The joining of package specifications is based on matching by the same name, or joining through explicit invariants. Individual role models are expected to result in crosscutting differences in practical applications. The differences would need to be addressed during integration. However the model does not specify any late aspect mechanism to address crosscutting concerns.

Subject-oriented programming (SOP) [40] represents a concept of decomposing a system into modules, that can be developed separate from one another, to be composed later into a complete system. The system can be decomposed from any perspective into separate implementation modules, called *subjects*. A subject defines some of the state and behaviour of objects in many classes; different subjects separately define and operate upon shared objects without needing to know the details associated with those objects by other subjects. A *subject compositor* tool is supposed to combine different subjects through some integration rules, to result in a fully functioning system. A symmetric composition of individual subjects is expected to suffice for a successful integration into a complete system. Whether difficulties arise in practice, or would a symmetric composition suffice in practice, has not been evaluated. SOP has been extended through various AOSD approaches like multi-dimensional separation of concerns [85], Theme [25], concern modelling of Cosmos [82], and tools like Hyper/J [84] and concern manipulation environment (CME) [41] tool suite of Concern Explorer for Eclipse, Concern Modeller, Hyper/J2,

HyperProbe etc.

The approach of multi-dimensional separation of concerns [85] extends SOP, through proposing a decomposition process based on separate dimensional concerns that are non-orthogonal to one another. Each concern is encapsulated into a separate *hyperslice* that contains modules with only those entities that pertain to the encapsulated concern. Separate hyperslices are composed, according to a set of rules into a *hypermodule*, which can be considered as a single hyperslice. A final hypermodule, through integration of all different hyperslices, can thus represent the complete system. The composition technique identifies matching units in different hyperslices, reconciles differences, and integrates all the units to produce a unified whole. However, the approach does not prescribe any specific decomposition technique to derive the hyperslices, and neither does it specify any formalism to define the composition technique.

Concern-space modelling schema, Cosmos [83] complements the concept of SOP, to provide advanced separation of concerns across the lifecycle work-products. Cosmos represents concern spaces in terms of concerns, relationships, and predicates. Concerns are categorized as logical, representing conceptual considerations; and physical, representing the actual elements of software systems. A study by Sutton and Rouvellou [82] shows application of the Cosmos model in conjunction with Hyper/J [84] to provide a uniform separation of concerns across the software lifecycle. The model can be treated as complementary to a symmetric modelling approach like multi-dimensional separation of concerns, to provide explicit mapping and correlation of concerns across different stages of the lifecycle; however, development and integration of concerns have not been prescribed explicitly.

The different AOSD work based on the symmetric approach, have promoted different means for decomposing a system that can provide better "separation of concerns" compared to OO technique. It is expected that the separate design model encapsulating individual concerns, would differ among their respective perspectives on the system. These differences are ideally crosscutting concerns that need to be reconciled to integrate the individual design models into a complete system. Most of the symmetric AOSD models have partially or incompletely addressed the explicit mechanism to address crosscutting concerns late in the lifecycle (during integration).

Some AOSD work, not following the symmetric modelling of concerns, have also promoted the late aspect approach. Work on implicit context by Walker and Murphy [93, 92], proposes an alternative means to support separation of concerns, not following the typical symmetric AOSD. The approach considers a decomposition of the system into components; each component can possess a localized, inconsistent view of its context of operation, avoiding any *extraneous knowledge* (of the outside world). Differences of the local and global perspectives are resolved by addressing the crosscutting differences during integration (through "contextual dispatch") of the components into a complete system.

Work by Rajan and Sullivan [68, 69] addresses separation of integration concerns as aspects. Component interactions during integration are reconciled by aspects that act as mediators. The proposed tool Eos, also provides a rich join point model to address the complex interactions during integration. Work by Kande and Strohmeier [46] proposes an extension to UML, to model software architecture with a connector that encapsulates complex component interactions. Each component considers the different architectural perspectives on the connector; this aims at pro-

viding improved separation of concerns. The crosscutting interactions among different system components are reconciled by the connector (during integration); each component has to adhere to the architecture of the connector. Implication of such architectural dependency on software evolution, is yet to be addressed.

7.3 Empirical Evaluations of AOSD

There has been some work to evaluate a number of AOSD models and tools. This section discusses some of the notable ones.

Walker and colleagues [94] performed two semi-controlled experiments that considered whether the effects of explicit separation of crosscutting concerns were beneficial or harmful in working on two common programming tasks: debugging and change. The results of the experiments were mixed. One demonstrated that adoption of separation of aspects helped the participants completing debugging tasks faster, compared to the participants using traditional OO technique; the other one demonstrated that the participants adopting separation of the crosscutting concerns, required relatively more time in incorporating changes. Baniassad and colleagues [12] performed similar studies later in an industrial context evaluating complexity of identifying and changing tasks involving crosscutting concerns. The results show that non-separated crosscutting concerns represent an impediment to software evolution tasks in practice. Murphy and colleagues [62] provide some initial evaluation of the abilities of AspectJ and Hyper/J tools to refactor crosscutting concerns in existing systems; the symmetric modelling of the crosscutting concerns in Hyper/J was found to be more beneficial under some circumstances. Unfortunately, the Hyper/J

tool remained a partially realized, proof-of-concept implementation unsuitable for industrial use. Thang and Katayama also considered Hyper/J to be beneficial for implementing collaboration-based designs relative to Java and C++-templates [87].

Various case studies have been conducted in the aspect-oriented implementation of systems (e.g., Kersten and Murphy [49]). Hannemann and Kiczales [39] demonstrate how the implementations of certain design patterns can be separated via AspectJ. Kienzle and Guerraoui [52] argue that separation of crosscutting concerns can be harmful under some situations, reinforcing earlier findings. Coady and Kiczales [29] have investigated AspectJ-style separation of some crosscutting concerns in an operating system, and analyzed the effects of this separation on the re-visited evolution history of that system. The study considered evolution of specific concerns only, not considering complete successive versions of the system; the results indicate improvement of modularity, and extensibility for the separated out aspects. Siadat and Walker [77] investigated the effects of separation of optimization features for a network simulator, as explicit crosscutting concerns. The study shows that explicit separation of the crosscutting features provides modularization without degrading performance. The study also indicates concerns regarding comprehensibility of the system as the aspects required fine-grained details of the base. Tsang and colleagues [88] evaluated usefulness of separation of concerns based on C&K metric suite [19], by applying aspect-oriented programming with AspectJ in development of real-time systems. The key metrics that were used to compare between aspect-oriented and object-oriented versions were: depth of inheritance tree, weighted methods per class, coupling between objects, and lack of cohesion among methods. Their results suggest that separation of concerns can improve modularity,

and understandability.

7.4 Summary

Based on the positive evidences in applying asymmetric AOSD, early aspect approaches assume that the early separation of crosscutting concerns should be able to better address software development and evolution. Similarly, a number of AOSD techniques have promoted the late aspect concept, expecting that the late treatment of crosscutting concerns can better address software evolution. However, whether either approach can provide a better solution to address software development and evolution in practice, remains unexplored.

Harrison *et al.* [42] provides a comparative analysis between symmetric and asymmetric approaches, where they argue that asymmetric modelling can be useful for co-evolving crosscutting concerns, while symmetric modelling can provide better support to component reuse and evolution. Arguably, early aspect approaches should further improve the results for asymmetric AOSD, and late aspect approaches should be able to provide better support to symmetric AOSD. Although it has been realized that crosscutting concerns need to be addressed to provide support for software evolution, it is yet to be determined, when in the lifecycle crosscutting concerns should be treated. An evaluation of early and late aspect approaches should be valuable in addressing this question.

Chapter 8

Conclusion

Non-modularity of crosscutting concerns has been identified as a key impediment for object-oriented technology to address software evolution. Aspect-oriented technologies, by addressing crosscutting concerns, aim at providing support for the MC properties (traceability, comprehensibility, independent development, and evolvability), important to address evolution. Two key ideals for addressing crosscutting concerns across the software development lifecycle, the early and the late aspect approach, have been promoted through different AOSD work. Early aspect approaches, based on asymmetric AOSD, consider identification and separation of crosscutting concerns from the base early in the lifecycle; modularization of crosscutting concerns throughout the lifecycle is expected to provide support for the MC properties and thereby, address software evolution. Late aspect approaches do not consider separation of crosscutting concerns from the base until the design, rather crosscutting concerns are only addressed during the integration phase; late treatment of crosscutting concerns is expected to be useful in addressing the software properties, important to evolution.

Evolution is considered an essential aspect of software systems, as all systems evolve through their life-times; if achievable in a feasible means, the software properties important to evolution are valuable to software engineering. Different AOSD techniques, introduced based on either of the concepts of early and late aspects, claim to address the important properties to support software evolution. However,

with most AOSD research focussing on means to apply the early or the late aspect approach, the key questions of whether either (or both) of the concepts can provide a solution to address the valuable software properties (related to evolution) in practice, would one be better or more feasible over the other, or should one be preferred over the other in certain situations, remain unaddressed. An evaluation of early and late aspect approaches with respect to the MC properties, can address the key issues and hence, can be significant to address the general question of, when in the lifecycle crosscutting concerns should be addressed.

In this thesis, we provide an initial evaluation of early and late aspect approaches by evaluating an AOSD model, Theme, which has been promoted via both the approaches. The early aspect model of Theme can be considered as one of the better early aspect approaches to address software evolution; most early aspect approaches fail to address evolution, focussing only on means to identify crosscutting concerns. The late aspect model of Theme can be considered as a representative late aspect approach for symmetric AOSD. Evaluation of the two models of Theme is presumed to represent an evaluation of the underlying approaches of early and late aspects.

We applied both the Theme models in the development and evolution of a benchmark system. We selected FTP server as the benchmark system, as it proved to be of reasonable size and complexity to investigate the MC properties. We applied both the models on the identical setting and developed a base system. We investigated the MC properties of traceability, comprehensibility, and independent development along the development process. We then applied several evolution steps on the base system to investigate the evolvability property. Each version was developed without consideration of the future changes, in order to investigate how the models fared to

support unanticipated evolution.

The results of the study demonstrate that the early aspect model of Theme fails to address independent development and evolvability. The early separation of crosscutting concerns did not suffice for a successful integration of independent design models into a complete system; as a result, independent development of the system cannot be supported by the early aspect model of Theme. With an evolution step introducing new crosscutting concerns, we found that non-invasive evolution with the early aspect model could only be ensured at the cost of breaking the asymmetric separation of the system. Since a key principle of the early aspect approach is to maintain the clean asymmetric separation between the base and crosscutting concerns, evolvability cannot be addressed on the part of the early aspect model. The late aspect model of Theme on the other hand, proved to provide support for both independent development and evolvability, through addressing crosscutting concerns during integration. Late addressing of crosscutting concerns sufficed for successful integration of independent design models; new design models, encapsulating new requirements, could also be incorporated into the existing system in an additive manner.

Both the Theme models proved to support traceability for the system requirements, from the specification document to the design and implementation artifacts. However, if the early aspect model does not consider processing of all the system requirements during an evolution step (in order to preserve asymmetric modelling), traceability of crosscutting requirements may not hold; to ensure traceability for a crosscutting requirement, an asymmetric separation between the base and the crosscutting concerns needs to be maintained throughout the lifecycle. Comprehen-

sibility proved to be elusive with either of the Theme models. Considering that a maintenance team would only have to deal with one design model for a particular evolution task, improved comprehensibility should be ensured in case of both the models. However, design for an aspect theme (in the early aspect model) can get complex as it requires detailed knowledge about multiple design themes. Identification of the crosscutting differences during integration, on the part of the late aspect model, can also require detailed understanding of individual design themes, affecting comprehensibility.

The study provides an initial evaluation of early and late aspect approaches in general, with respect to the properties important to software evolution. The early aspect model of Theme can be considered as one of the better early aspect approaches that it addresses software evolution explicitly. Most early aspect approaches fail to address non-invasive evolution. The results from applying the early aspect model of Theme in evolution of the system, indicates that the problem lies in the early aspect concept (of asymmetric separation of crosscutting concerns early in the lifecycle) itself; work towards improved means to identify, or represent crosscutting concerns (on the part of an early aspect approach), can at best provide a better modularized means for developing the base system; but the issues with evolution of the system would remain unresolved. The late aspect model of Theme represents a late aspect approach to support a symmetric AOSD model. The results of the study indicate that consideration of crosscutting concerns during the integration phase, can lead symmetric AOSD towards addressing software evolution in a non-invasive manner. The study however, does not represent evaluation of late aspect approaches (with respect to the MC properties) that do not consider symmetric AOSD.

The main contribution of the thesis is providing an evaluation of two AOSD approaches with respect to software evolution. The evaluation provides an initial assessment of a significant software engineering question, when we should address crosscutting concerns in the lifecycle. The evaluation process also explores the lifecycle issues of an AOSD model, Theme, determining the feasible means for applying the model in the development and evolution of software systems.

Results of the study indicate that crosscutting concerns should be addressed late in the lifecycle to support software evolution. This initial evaluation should motivate future work on evaluating late aspect approaches, through their applications on large scale industrial systems; future evaluations should also consider late aspect approaches that do not consider symmetric AOSD, to provide comprehensive comparative analysis of late aspects.

Evaluation of late aspect approaches, in addressing reuse of sub-systems (or features) among unrelated software systems, should also be interesting future work; the initial evaluation indicates that the late consideration of crosscutting concerns may address unanticipated software reuse. The work should also motivate future research on addressing crosscutting concerns during integration, in a more comprehensive manner. It is not clear with current late aspect approaches, how much work it would be to identify and address crosscutting differences among the parts of a large, complex system, developed independently.

Bibliography

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods. In *e-ICOLC*, 2002.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Workshop on Object-Based Distributed Programming ECOOP'93*, 1994.

[3] S. W. Ambler. *Agile Modeling*. John Wiley & Sons, 1st edition, May 2002.

[4] J. Araujo and P. Coutinho. Identifying aspectual use cases using a viewpoint-oriented requirements method. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2003.

[5] J. Araujo and A. Moreira. An aspectual use case driven approach. In *VIII Jornadas de Ingeniera de Software Bases de Datos (JISBD)*, 2003.

[6] J. Araujo, A. Moreira, I. Brito, and R. Rashid. Aspect-oriented requirements with uml. In *Workshop on Aspect-Oriented Modelling with UML. UML'02*, 2002.

[7] J. Araujo, J. Whittle, and K. Dae-Kyoo. Modeling and composing scenario-based requirements with aspects. In *International Conference on Requirements Engineering*, 2004.

[8] J. Bakker, B. Tekinerdoan, and A. Aksit. Characterization of early aspects approaches. In *Early Aspects 2005: Aspect-Oriented Requirements Engineering and Architecture Design*, 2005.

[9] J. Banatre and D. Metayer. Programming by multiset transformation. *Communications of ACM*, 36(1), 1993.

[10] E.L.A. Baniassad and S. Clarke. Finding aspects in requirements with theme/doc. In *Early Aspects Workshop, AOSD'04*, 2004.

[11] E.L.A. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *International Conference on Software Engineering*, 2004.

[12] E.L.A. Baniassad, G.C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Internation Conference on Aspect-Oriented Software Development*, 2002.

[13] D. Barritt. Dsdm in real-time process control application. *IEE Computing and Control*, pages 94–100, April 2002.

[14] K. Beck. Embracing change with extreme programming. *Computer*, pages 70–77, October 1999.

[15] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[16] L. Belady and M. Lehman. A model of large program development. *IBM Systems*, 15(3), 1973.

[17] B. Boehm. A spiral model for software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[18] F. Brooke. *Mythical Man-Month, The: Essays on Software Engineering*. Addison-Wesley, 1957.

[19] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Soft. Eng.*, 1994.

[20] R. Chitchyan, A. Rashid, and P. Sawyer. Comparing requirements engineering approaches for handling crosscutting concerns. In *Workshop on Requirements Engineering, CAiSE*, 2005.

[21] S. Ciraci and P. Broek. Evolvability as a quality attribute of software architectures. In *International ERCIM Workshop on Software Evolution'06.*, 2006.

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

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

[24] S. Clarke and E. Baniasad. *Aspect-oriented Analysis and Design: The Theme Approach*. Object Technology. Addision-Wesley, first edition, March 2005.

[25] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design. In *ACM Conferemce on Object-Oriented Programming, Language, Systement, and Applications*, 1999.

[26] S. Clarke and R. Walker. Towards a standard design language for AOSD. In *International Conference on Aspect-Oriented Software Development*, 2002.

[27] S. Clarke and R.J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, 2001.

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

[29] Y. Coady and G. Kiczales. A retroactive study of aspect evolution in operating system code. In *International Conference on Aspect-Oriented Software Development*, 2003.

[30] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *International Workshop on Principles of Software Evolution*, 2001.

[31] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[32] D. D'Souza and A. Wills. Modeling components and frameworks with uml. *Communications of ACM*, 42(10), 2000.

[33] C. Duan, C. Cleland-Huang, R. Settimi, and X. Zou. Decision making support for early aspects identification based on probabilistic trace retrieval. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2005.

[34] T. Elrad, M. Aksit, G. Kiczales, C. Lieberherr, and H. Ossher. Discussing aspects of aop. *Communications of ACM*, 44(10), 2001.

[35] B. Gaines and M. Shaw. Webgrid: Knowledge modeling and inference through the world wide web. Technical report, Knowledge Science Institute, University of Calgary,, 1993.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1994.

[37] J. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *International Conference on Requirements Engineering*, 1999.

[38] J. Grundy. Supporting aspect-oriented component based systems engineering. In *International Coference on Software Engineering and Knowledge Engineering*, 1999.

[39] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Language, and Applications*, 2002.

[40] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications*, 1993.

[41] W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Supporting aspect-oriented software development with the concern manipulation environment. *IBM Systems*, 44(2), 2005.

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

[43] R. Helm, I. Holland, and D. Gangopadhay. Contracts: Specifying behavioral composition in object-oriented systems. In *OOPSLA*, 1990.

[44] I. Holland. Specifying reusable components using contracts. In *European Conference on Object-Oriented Programming (ECOOP)*, 1992.

[45] I. Jacobson, I. Chirsterson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[46] M. M. Kande and A. Strohmeier. Modeling crosscutting concerns using software connectors. In *OOPSLA'2001 Workshop on Advanced Separation of Concerns*, 2001.

[47] S. Katz and R. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *International Conference on Requirements Engineering*, 2004.

[48] S. Katz and R. Rashid. Probe: From requirements and design to proof obligations for aspect-oriented systems. Technical report, Computing Department, Lancaster University, Lancaster, 2004.

[49] M. Kersten and G. Murphy. Atlas: A case study. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications*, 1999.

[50] Des Rivires J. Bobrow D. Kiczales, G. The art of the meta-object protocol. *MIT Press*, 1991.

[51] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 1997. LNCS 1241.

[52] J. Kienzle and R. Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *European Conference on Object-Oriented Programming*, 2002.

[53] G. Kotonya and I. Sommerville. *Requirements Engineering.* John Wiley & Sons, 2002.

[54] B. Kristensen. Subject composition by roles. In *Object-Oriented Information Systems (OOIS)*, 1997.

[55] K. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based customization. *Commun. ACM*, 37(5), 1994.

[56] C. Lott. Breathing new life into the waterfall model. *IEEE Software*, pages 103–105, September 1997.

[57] R. Medachy. *A software project dynamics model for process cost, schedule and risk assessment.* PhD thesis, University of Southern California,, 1994.

[58] K. Mens. Architectural aspects. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002.

[59] S. Miller, R. DeCarlo, and A. Mathur. Modeling and control of the incremental software test process. In *Computer Software and Applications Conference, COMSAQ'04*, 2004.

[60] A. Moreira, J. Araujo, and I. Britto. Crosscutting quality attributes of requirements engineering. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002.

[61] M. Mousavi, G. Russelo, M. Chaudron, M. Raniers, T. Basten, A. Corsaro, S Shukla, R. Gupta, and D. Schmidth. Aspects + gamms = aspectgamma: A formal framework for aspect-oriented specification. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2002.

[62] G.C. Murphy, A. Lai, R.J. Walker, and M.P. Robillard. Separating features in source code: An exploratory study. In *International Conference on Software Engineering*, 2001.

[63] H. Ossher et al. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3), 1996.

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

[65] David Parnas. Software aging. In *International Conference on Software Engineering*, 1994.

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

[67] R. Pressman. *Software Engineering: A Practitioner's Approach*, volume 1. McGraw-Hill Series in Software Engineering and Technology, 1955.

[68] H. Rajan. One more step in the direction of modularized integration concerns. In *International Conference on Software Engineering*, 2005.

[69] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *International Conference on Software Engineering*, 2005.

[70] R. Rashid, A. Moreira, and J. Arajo. Modularisation and composition of aspectual requirements. In *International Conference on Aspect-Oriented Software Development*, 2003.

[71] R. Rashid, A. Moreira, J. Arajo, and A. Sampaio. A multi-dimensional, model-driven approach to concern identification and traceability.

[72] R. Rashid, P. Sawyer, A. Moreira, and J. Arajo. Early aspects: a model for aspect-oriented requirements engineering. In *Joint International Conference on Requirements Engineering*, 2002.

[73] T. Reenskaug, P. Wold, and O. Lehne. *Working with objects. The OOram Software Engineering Method.* Manning Publications Co., 1995.

[74] D. Reifer, F. Maurer, and H. Erdogmus. Scaling agile methods. *IEEE Software*, pages 12–14, July 2004.

[75] A. Sampaio, N. Loughran, A. Rashid, and P. Rayson. Mining aspects in requirements. In *Workshop on Early Aspects, AOSD'05*, 2005.

[76] J. Shilling and P. Sweeny. Three steps to views: Extending the object-oriented paradigm. In *OOPSLA*, 1989.

[77] J. Siadat, R. Walker, and C. Kiddle. Optimization aspects in network simulation. Technical report, Dept. of Computer Science. University of Calgary, 2005.

[78] J. Sidoran, C. Burns, S. Maethner, D. Spencer, and H. Bond. A case study on rapid systems prototyping and its impact on system evolution. In *IEEE International Workshop on Rapid Prototyping*, 1995.

[79] I. Sommerville and P. Sawyer. Preview viewpoints for process and requirements analysis. Technical report, Lancaster University, Lancaster, 1996.

[80] A. Steed and J. McDonnell. Experiences with repertory grid analysis for investigating effectiveness of virtual environments. Technical report, Department of Computer Science, University College London, UK,, 2003.

[81] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for aspectj. In *International. Conference on Aspect-Oriented Software Development*, 2002.

[82] S. Sutton and R. Isabelle. Advanced separation of concerns for component evolution. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Language, and Applications*, 2001.

[83] S. Sutton and P. Tarr. Early-stage concern modeling in general. In *International Conference on Aspect-Oriented Software Development*, 2002.

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

[85] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, 1999.

[86] R. Taylor, N. Medvidvovic, K. Anderson, Whitehead J., J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A component- and message-based architectural style for gui software. *IEEE Trans. Soft.*, 22(6), 1996.

[87] N.T. Thang and T. Katayama. Collaboration-based evolvable software implementations: Java and Hyper/J vs. C++-templates composition. In *International Workshop on Principles of Software Evolution*, 2002.

[88] S. Tsang and E. Clarke, S. amd Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. Technical report, Department of Computer Science, Trinity College Dublin,, 2003.

[89] C. Turner. *Feature Engineering of Software Systems.* PhD thesis, Dept. of Computer Science. University of Colorado., 1999.

[90] C. Turner, A. Fuggeta, L. Lavazza, and A. Wolf. A comceptual basis for feature engineering. *Systems and Software*, 1999.

[91] D. Wagelaar. A concept-based approach for early aspect modelling. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2003.

[92] R. Walker. Supporting inconsistent world views. In *International Conference on Aspect-Oriented Softare Development*, 2003.

[93] R. Walker and G. Murphy. Implicit con text: Easing software evolution and reuse. In *Proceedings of the ACM SIGSOFT. FSE-8*, 2000.

[94] R.J. Walker, E.L.A. Baniassad, and G.C. Murphy. An initial assessment of aspect-oriented programming. In *International Conference on Software Engineering*, 1999.

[95] R.J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004. To appear.

[96] J. Whittle and J. Araujo. Scenario modeling with aspects. *IEE Proceedings-Software*, 151, 2004.

[97] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering*, 2000.

[98] L Xu, H. Ziv, and D. Richardson. Towards modeling non-functional requirements in software architecture. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2005.

[99] Y. Yu, J. Leite, and J. Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *International Conference on Requirements Engineering*, 2004.

Appendix A

A discussion on the structured requirements specification

In this appendix, we present the structured requirements specification, we derived in applying the early aspect model of Theme, and also discuss a sample repertory grid we used to validate the structured set of requirements. Next, we present the new set of requirements to evolve the base FTP system into Version 2, and discuss mapping among the new set of requirements and the themes with the Theme/Doc process.

A.1 Requirements for minimum implementation of FTP Server

R1 The server shall listen at port L.

R2 The server shall open a control connection at the request of Client process.

R3 Upon opening connection with a user, the server shall send the following Reply Message:

220 + *corresponding message*

R4 The system shall interpret a user request and send either of the following Reply Messages:

421 + *corresponding message*

500 + *corresponding message*

User Name (user)

R5 The argument field is a Telnet string identifying the user.

R6 Server may allow a new USER command to be entered at any point in order to change the access control and/or accounting information. This has the effect of flushing any user information.

R7 According to the action performed, the system shall send the following associated reply messages:

230 + *corresponding message*

500 + *corresponding message*

Logout (quit)

R8 This command terminates a USER and if file transfer is not in progress, the server closes the control connection.

R9 If file transfer is in progress, the connection will remain open for result response and the server will then close it.

R10 Before closing Control Connection the system shall send the following Associated reply messages:

221 + *corresponding message*

R12 If file transfer in progress with the user then the system shall send the following reply message:

426 + *corresponding message*

Data Port (port)

R11 The argument is a HOST-PORT specification for the data port to be used in data connection.

R12 There are defaults for both the user and server data ports, and under normal circumstances this command and its reply are not needed.

R13 If this command is used, the argument is the concatenation of a 32-bit internet host address and a 16-bit TCP port address. This address information is broken into 8-bit fields and the value of each field is transmitted as a decimal number (in character string representation). The fields are separated by commas. A port command would be:

port h1,h2,h3,h4,p1,p2

where h1 is the high order 8 bits of the internet host address.

R14 If the argument has a valid data port syntax then the system store the user's port and send the following reply message:

200 + *corresponding message*

R15 If the argument is not a valid port syntax then the system shall send either of the following reply messages:

500 + *corresponding message*

501 + *corresponding message*

Representation Type (type)

R16 The argument specifies the representation type.

R17 Several types take a second parameter. The first parameter is denoted by a single Telnet character, as is the second Format parameter for ASCII and

EBCDIC; the second parameter for local byte is a decimal integer to indicate Bytesize. The parameters are separated by a <SP> (Space, ASCII code 32). The following codes are for type are supported by the "minimum implementation of FTP":

A - ASCII + N - Non-print

R17 The default representation type is ASCII Non-print. If the Format parameter is changed, and later just the first argument is changed, Format then returns to the Non-print default.

R18 If the argument is supported by the system's "minimum implementation version", it shall set the type and send the following reply message:

200 + *corresponding message*

R19 If the argument field is not supported by the version then the system shall send the following reply message:

504 + *corresponding message*

R20 If the argument field is not valid then the system shall send either of the following reply messages:

500 LineTooLongReply*

501 ArgumentSyntaxErrorReply

Transfer Mode (mode)

R21 The argument is a single Telnet character code specifying the data transfer modes.

R22 The following codes are assigned for transfer modes:

S - Stream

B - Block

C - Compressed

R23 The system shall support Stream as The default transfer mode.

R24 If the argument field is not valid then the system shall send either of the following reply messages:

500 + *corresponding message*

501 + *corresponding message*

R25 If the argument is supported by the system's "minimum implementation version", it shall set the type and send the following reply message:

200 + *corresponding message*

R26 If the argument field is not supported by the version then the system shall send the following reply message:

504 + *corresponding message*

File Structure (stru)

R27 The argument is a single Telnet character code specifying file structure.

R28 The following codes are assigned for structure:

F - File (no record structure)

R - Record structure

P - Page structure

R29 The system shall support File as the default structure.

R30 If the argument field is not valid then the system shall send either of the following reply messages:

500 + *corresponding message*

501 + *corresponding message*

R31 If the argument is supported by the system's "minimum implementation version", it shall set the type and send the following reply message:

200 + *corresponding message*

R32 If the argument field is not supported by the version then the system shall send the following reply message

504 + *corresponding message*

No operation (noop)

R33 This command does not affect any parameters or previously entered commands. It specifies no action other than that the server send reply message:

200 + *corresponding message*

Send file (retr)

R34 This command causes the server-DTP to transfer a copy of the file, specified in the pathname, to the server- or user-DTP at the other end of the data connection. The status and contents of the file at the server site shall be unaffected.

R35 The system shall send the following reply message and open a Data Connection

with the user:

150 + *corresponding message*

R36 If Data Connection is already open with the user then the system shall send the following reply message:

125 + *corresponding message*

R37 In case of failure to establish a Data Connection, the system shall send the following reply message:

425 + *corresponding message*

R38 If the argument field is invalid then the system shall send the following reply message:

500 LineTooLongReply*

501 + *corresponding message*

R39 If the file named according to the argument passed is not found then the system shall send either of the following reply messages:

450 + *corresponding message*

550 + *corresponding message*

R40 After transferring file the system shall send the following reply messages:

250 + *corresponding message*

226 + *corresponding message*

R41 Failure to complete file transfer will result in the following reply message:

451 + *corresponding message*

Receive File (stor)

R42 This command causes the server-DTP to accept the data transferred via the data connection and to store the data as a file at the server site.

R43 If the file specified in the pathname exists at the server site, then its contents shall be replaced by the data being transferred.

R44 A new file is created at the server site if the file specified in the pathname does not already exist.

R45 The system shall send the following reply message and open a Data Connection with the user:

150 + *corresponding message*

R46 If Data Connection is already open with the user then the system shall send the following reply message:

125+ *corresponding message*

R47 In case of failure to establish a Data Connection, the system shall send the following reply message:

425 + *corresponding message*

R48 If the argument field is invalid then the system shall send the following reply message:

500 + *corresponding message*

501 + *corresponding message*

R49 After transferring file the system shall send the following reply messages:

250 + *corresponding message*

226 + *corresponding message*

R50 Failure to complete file transfer will result in the following reply message:

451 + *corresponding message*

R51 All commands shall be processed under a separate session. Storing and retrieval off all information related to a particular user shall be processed with respect to a distinct session for the particular connection with that user.

***Reply messages*:**

A reply is an acknowledgment (positive or negative) sent to user via the control connection in response to FTP commands. The general form of a reply is a completion code (including error codes) followed by a text string. The codes are for use by programs and the text is usually intended for human users.

120 Service ready in nnn minutes.

125 Data connection already open; transfer starting.

150 File status okay; about to open data connection.

200 Command okay.

202 Command not implemented, superfluous at this site.

211 System status, or system help reply.

214 Help message. On how to use the server or the meaning of a particular non-standard command. This reply is useful only to the human user. 220 Service ready for new user.

221 Service closing control connection. Logged out if appropriate.

225 Data connection open; no transfer in progress.

226 Closing data connection. Requested file action successful (for example, file transfer or file abort).

227 Entering Passive Mode (h1,h2,h3,h4,p1,p2).

230 User logged in, proceed.

250 Requested file action okay, completed.

331 User name okay, need password.

332 Need account for login.

421 Service not available, closing control connection.

425 Can't open data connection.

426 Connection closed; transfer aborted.

450 Requested file action not taken. File unavailable (e.g., file busy).

451 Requested action aborted: local error in processing.

452 Requested action not taken. Insufficient storage space in system. 500 Syntax error, command unrecognized.

501 Syntax error in parameters or arguments.

502 Command not implemented.

503 Bad sequence of commands.

504 Command not implemented for that parameter.

530 Not logged in.

532 Need account for storing files.

550 Requested action not taken. File unavailable (e.g., file not found, no access).

552 Requested file action aborted.

553 Requested action not taken. File name not allowed.

A.1.1 The repertory grid

To verify whether the structured set of requirements validly represent the description of FTP server specified in RFC 959, we analyzed the requirements with Repertory Grid technique. We derived a set of grids with WebGrid(III) tool and analyzed each of the requirements with the grid properties to see if any requirement contradicts a grid property. The tool takes a set of domain elements and a set of constructs, which compares and relates the elements with a view to providing better understanding about the domain.

Figure A.1 shows a grid (in 'map' format), we used in the study to investigate a number of requirements. The grid relates a number of elements ('user', 'server', 'port', 'request', 'control connection', 'data connection', and 'reply') with multiple constructs. The constructs, 'single/multiple' and 'initialized once per user' relate the elements 'user', 'server', 'control connection', 'request', and 'reply' denoting that a single server would communicate with multiple users; a control connection would be initialized once per user; and, there can be multiple number or replies and requests between a user and the server. The 'listen on' and 'connects to' constructs relate 'user' and 'server' with 'port'. This grid (displayed as a PrinCom map) provides some basic relationships for some domain elements for an FTP server. We verified the requirements that dealt with one or more elements mapped in the grid; we could check whether any requirement contradicted with any of the concepts represented in the map.

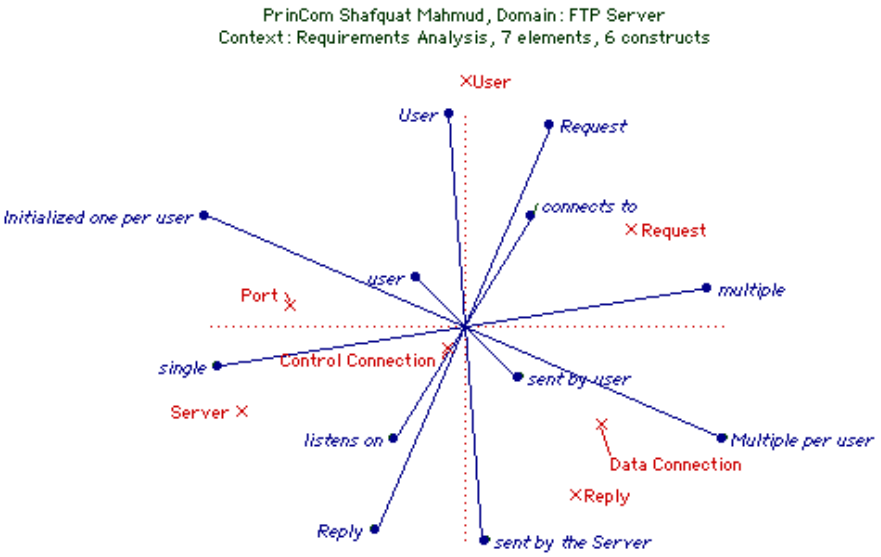

Figure A.1: A sample repertory grid showing relationship among FTP domain elements

A.2 Requirements for FTP Version 2

In this section, we first describe the structured set of requirements, that denote the new feature for Version 2 of the system. Next, we present the Theme/Doc mapping of the requirements with the initial themes, the refined (processed) set of requirements, and the final mapping of the requirements and the themes.

A.2.1 Initial set of new requirements

The following structured set of requirements represent the "user-password authentication" feature, we considered for the first evolution step.

User Name (user)

R53 The argument field is a Telnet string identifying the user.

R54 Server may allow a new USER command to be entered at any point in order to change the user information. This has the effect of flushing the name to store new user name, and beginning the login sequence again.

R55 According to the action performed, the system shall send the following Associated reply messages:
230 + *corresponding message* 241 + *corresponding message*

Password(pass)

R56 The argument field is a Telnet string specifying the user's password.

R57 This command must be immediately preceded by the user name command.

R58 If the argument supplied matches the corresponding user name, the system shall send the following reply message:

230 + *corresponding message*

R59 If the argument does not match the user-password, the system shall flush the current user name and send the following reply message:

241 + *corresponding message*

Verify Authentication

R60 The system shall verify authentication for certain FTP commands. If a user is not logged in with corresponding password (pass command), then the system shall not allow the FTP-commands from the client other than user, quit, and noop. Any unauthorized command request shall result in the following reply message:

241 + *corresponding message*.

A.2.2 The mapping among the new requirements and the themes

The initial mapping of the requirements with the themes (with the first two steps of Theme/Doc process), is shown in Figure A.2(a). Each requirement is shared among multiple themes. We next followed the Theme/Doc steps 3 to 7 to reconcile the sharing among the requirements and the themes, to provide a many to one mapping between the requirements and the themes. The final set of requirements after reconciliation, following the Theme/Doc process, is described below.

R53 The argument field is a Telnet string identifying the user.

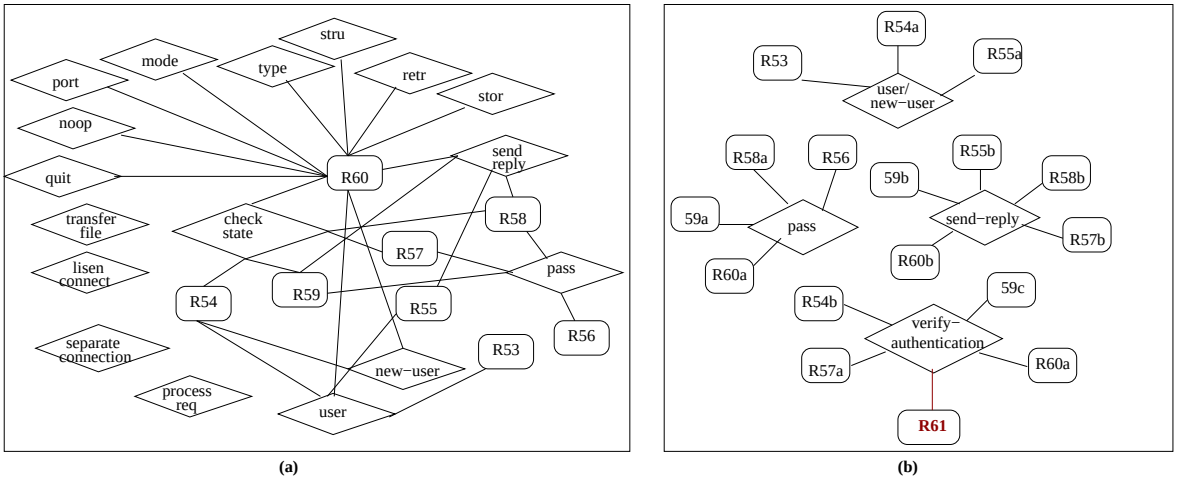

Figure A.2: Mapping among the new requirements and the themes

R54a Server may allow a new USER command to be entered at any point in order to change the access control and/or accounting information. This will flush any previous name and store name for the new user.

R54b For a new user, the system shall begin the login sequence again.

R55a Check validity of the user name.

R55b According to the validity of the user-name, the system shall send the following Associated reply messages:

330 + *corresponding message* 530 + *corresponding message*

R56 The argument field is a Telnet string specifying the user's password.

R57a This command must be immediately preceded by the user name command.

R57b For invalid sequence, send the following reply message: 503 + *corresponding message*

R58a Check if the argument supplied matches the corresponding user name.

R58b For a success in matching, the system shall send the following reply message respectively:

230 + *corresponding message*

R59a If the argument does not match the user-password, the system shall restart the log-in sequence.

R59b Send the following reply message in case of a "no-match":

530 + *corresponding message*

59c For a correct user-password match, update the machine state.

R60a The system shall verify authentication for certain FTP commands. If a user is not logged in with corresponding password (pass command), then the system shall not allow any other commands from the client other than user, quit, and noop.

R60b Any unauthorized command request shall result in the following reply message:

530 + *corresponding message*.

The following requirement went missing by the analysis process with Theme/Doc.

Added requirement:

[R61] Authentication verification for FTP-commands requested from a user, should be processed with respect to the specific connection between that user and the server.

Figure A.2(b) shows the mapping of the final set of requirements and the themes. No requirement is shared among two themes. The two new themes, `pass` and `new_user` qualified as base themes, and the theme, `verify_authentication` qualified as an aspect theme through the Theme/Doc process.

Appendix B

Design details for a number of themes

This appendix discusses design details for a number of themes and issues with their integration.

B.1 Themes along Path 1 with the early aspect model

In this section, we provide design details for a number of base and aspect themes (developed with the early aspect model of Theme) considering development of themes all at one time, according to Path 1. It can be expected that the themes developed together would conform to a common design architecture. There would be a uniform (defined) communication protocol whereby different themes can interact with one another, and it should be known to a theme, the specific means by which any particular theme can be triggered.

Figure B.1 shows two base themes, `user`, and `port` that encapsulate the features corresponding to the FTP commands, *user* and *port* respectively. Each theme was designed similar to Command design pattern [36]; a concrete `Command class` implements the behaviour corresponding to a particular FTP-command. Since the themes would perform similar behaviour whenever triggered, design principle of Flyweight design pattern [36] was followed to manage the references to the instance of a concrete `Command class`. As Figure B.1 shows, a theme encapsulating the feature corresponding to an FTP command, only has the view of the Flyweight

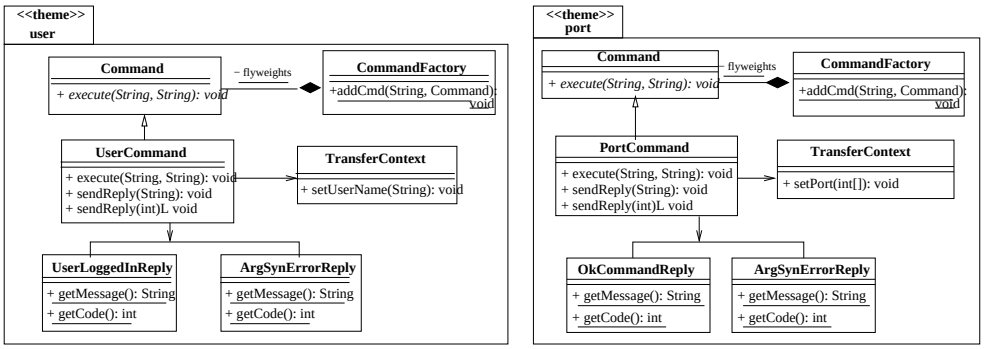

Figure B.1: Design of 2 base themes: "user" and "port"

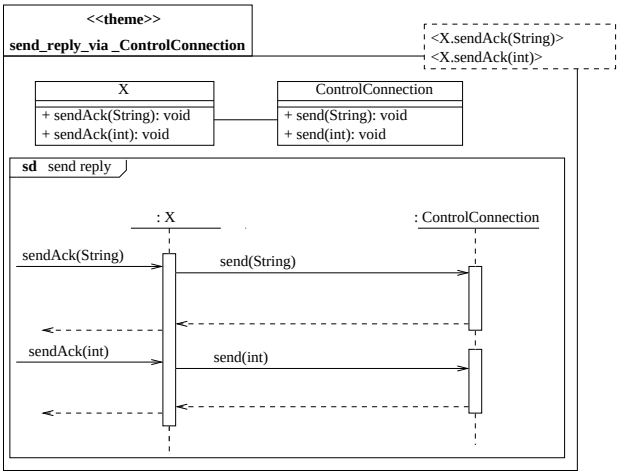

Figure B.2: Aspect theme: `send_reply_via_ControlConnection`

Factory (e.g. `CommandFactory class`) with the `Factory.add(..)` operation. Any other theme that would need to communicate with the `Flyweights`, would have the view of the Flyweight Factory with the getter operation (e.g. `public Command CommandFacotry.get(String)`). The design ensured that each base theme, encapsulating the feature corresponding to an FTP-command, would be triggered in the same manner; the `process_user_request` theme, which was responsible for processing the user request to interpret the corresponding FTP-command, could trigger the corresponding theme in the expected manner. It can also be observed that the base themes do not possess any view of the separated out crosscutting concerns, which are encapsulated into separate aspect themes. For example, references to an operation of an instance of `TransferContext class` should be specific to the particular connection with a user. But the base themes do no consider the details since, this crosscutting behaviour would be implemented by the corresponding aspect theme. Again, the themes do not consider the details about how a 'send-reply' operation would be performed, as the behaviour has also been separated out as a crosscutting one.

Figure B.2 shows a design for the aspect theme `send_reply_via_ControlConnection`. The theme is triggered by any request to send a reply-message to the user. As the sequence diagram shows, the request is replaced by a corresponding *send* operation of the ControlConnection class. Figure B.3 shows a more complex aspect theme, `perform_under_separate_connection`. The theme encapsulates the crosscutting behaviour of processing a particular user request with respect to a specific (separate) client-server connection. Below, we explain the three sequence diagrams shown in the figure.

The first sequence diagram(sd-accessUserInfo) shows how a getter or setter operation (retrieval or storing) regarding information of a user, referenced from an instance of a concrete Command class, is replaced with the corresponding operation of an instance of TrContext class; the instance should correspond to the specific connection between a user and the server. The second sequence diagram (sd-setConnection) shows how an instance of a concrete Command class is passed the appropriate value of the particular user-connection; the value is captured from the control flow path corresponding to the particular connection (control connection). The third sequence diagram(sd-resolveDataConn), similar to the first one, shows how any call to an operation of a DataConnection class, is replaced with a corresponding call to the appropriate (respective to the particular user-connection) instance of the class.

Figure B.4 shows the third aspect theme, transfer_file_via_DataConnection. The template parameters bind any request to transfer a file, and replace it by the corresponding operation of DataConnection class. It can be observed here that the aspect theme limits its scope within the description of the encapsulating feature, not addressing any other crosscutting behaviour. For example, the theme does not consider establishment, or usage of a 'data connection' with a particular user, since the crosscutting behaviour of, 'a user specific execution', has been implemented by a separate aspect theme (process_under_separate_connection). Transfer_file_via_DataConnection theme, in its execution, triggers the process_under_separate_connection aspect theme for the crosscutting behaviour.

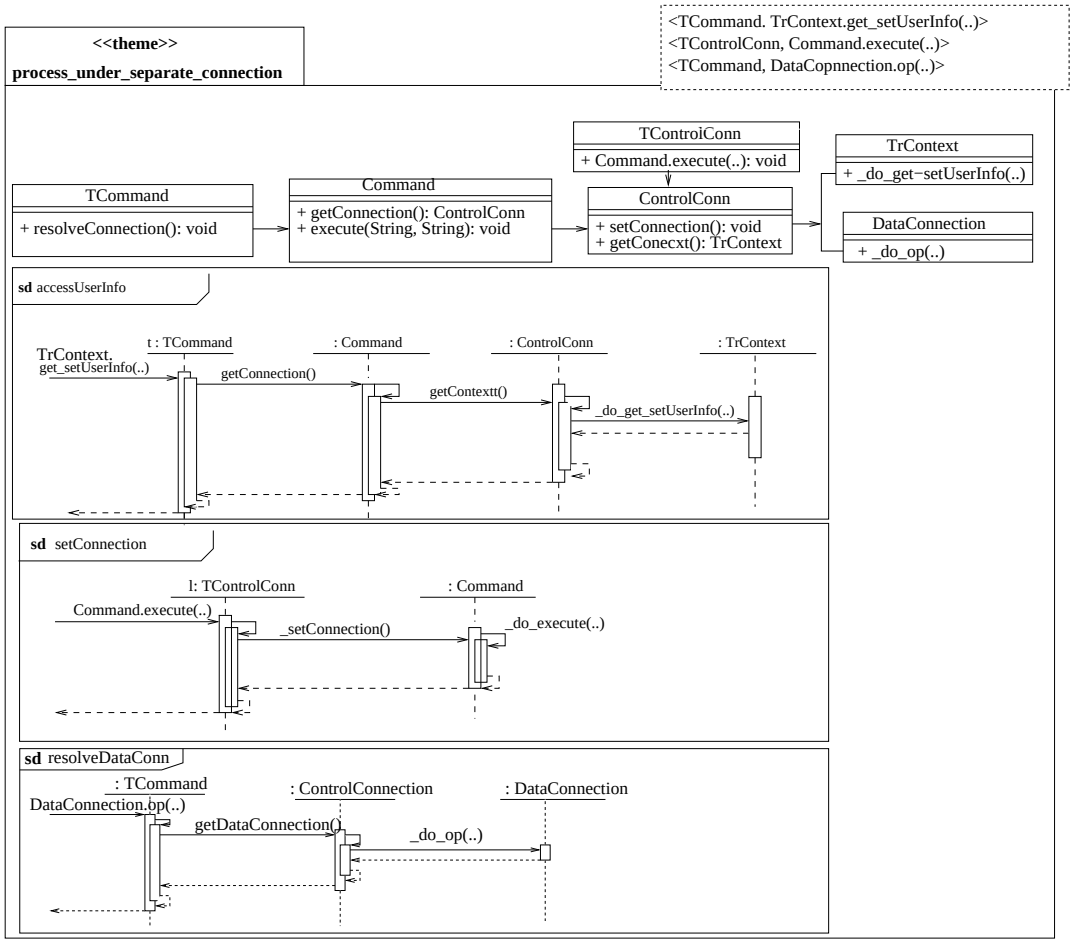

Figure B.3: Aspect theme: `process_under_separate_connection`

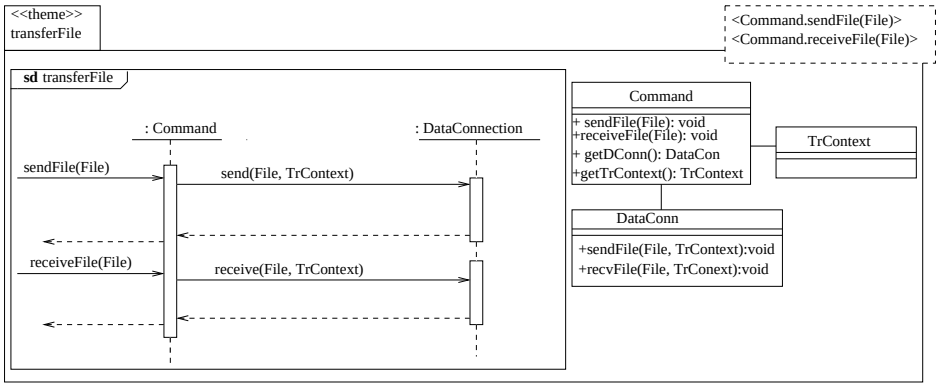

Figure B.4: Aspect theme: `transfer_file_via_DataConnection`

The themes, considered to be developed all at one time, conformed to a defined communication protocol. As a result, it was ensured that each theme would be triggered in a known manner. With the explicit knowledge about the design details of the base themes, as well as other aspect themes, an aspect theme could implement the crosscutting behaviour that would be triggered by other themes. Hence, integration of the themes did not result in any difference among interactions of different themes. Moreover, no conflicts occurred among the views of different themes, since they were considered to be developed together.

Thus, themes developed along the path, resulted in a successful integration into a complete system (Version 1).

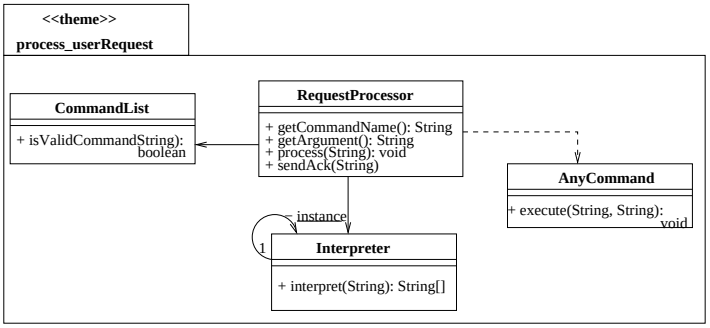

Figure B.5: `Process_user_requests` theme

B.2 Communication differences with early aspect model (Path 2.1)

This section describes communication differences with early aspect model along Path 2.1, by discussing composition of a number of base themes. Figure B.5 shows a design for `process_user_requests` theme along Path 2.1, with the early aspect model of Theme. `RequestProcessor.process(String)` method first interprets a user request and invokes `AnyCommand.execute(String, String)` method for execution of an FTP-command. `AnyCommand` class here is a representative of any class that would implement an FTP-command behaviour. Its composition with a theme, that encapsulates an FTP-command, is expected to provide a correct correspondence here. Figure B.6 shows designs for a number of themes along this path, as discussed in Chapter 4. Now let us consider integration of `process_user_requests` theme with `user` theme (in Figure B.6). The `user` theme, not possessing any view of the `RequestProcessor` class would result in a communication mismatch. `UserCommand`

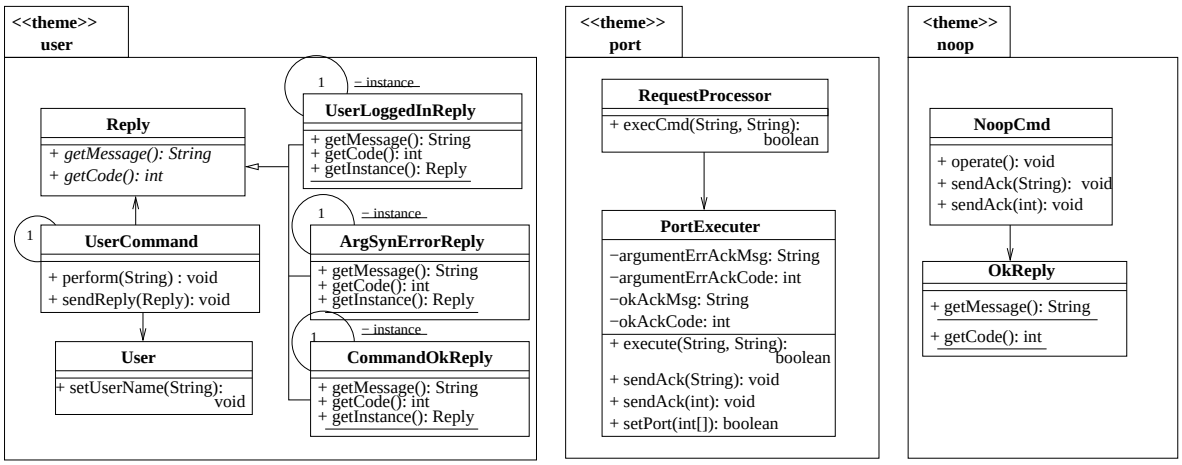

Figure B.6: Alternative designs for the base themes

class can override `AnyCommand` class during composition, but that would not resolve the communication differences. Without the explicit knowledge about how to communicate with `user` theme (e.g., by invoking the `perform(String)` operation of an instance of `UserCommand` class), design for the `process_user_requests` theme would not suffice for a successful composition.

`Port` theme in Figure B.6 has a view of `RequestProcessor` class. But integration of the theme with `process_user_requests` would still result in conflicts, as the method signatures do not match between the `RequestProcessor` classes. Similar problems arise in composing `process_user_requests` theme with `noop` theme.

We realized that the communication differences among individual themes need to be reconciled, in order to provide a successful integration with the early aspect model. However, the early aspect approach dies not consider reconciliation of crosscutting

differences after design. As a result, the path proved to be an infeasible one to result in a successful integration.

B.3 Themes along Path 1 with the late aspect model

According to the late aspect model of Theme, crosscutting concerns are not identified or separated out, rather they are considered implicitly from within individual design themes. In following Path 1 with the late aspect model of Theme, we considered design for individual themes conforming to a defined design architecture, with each theme encapsulating the corresponding feature irrespective of whether there was any embedded crosscutting concern or not. In this section, we demonstrate designs of a couple of themes from our study and discuss the feasibility of integration of the design themes along this path.

Figure B.7 shows designs of two themes, `user` and `port` along Path 1. The themes, considered to be designed together, followed a common design architecture; it was known to a theme how any particular theme expected to be communicated with. We can also see from the figure that, in encapsulating the corresponding features the themes have implicitly addressed concerns, which would have been identified and separated out as crosscutting ones in the early aspect approach. For example, each theme has a view of the `ControlConnection` class with respect to sending of reply-messages to the user. Each theme also considers processing of a user request with respect to the specific connection (with its view on `Session` class). Consideration of the shared (crosscutting) system behaviour on part of multiple individual themes, might lead to overlap or redundancy of work. We also had problems regarding

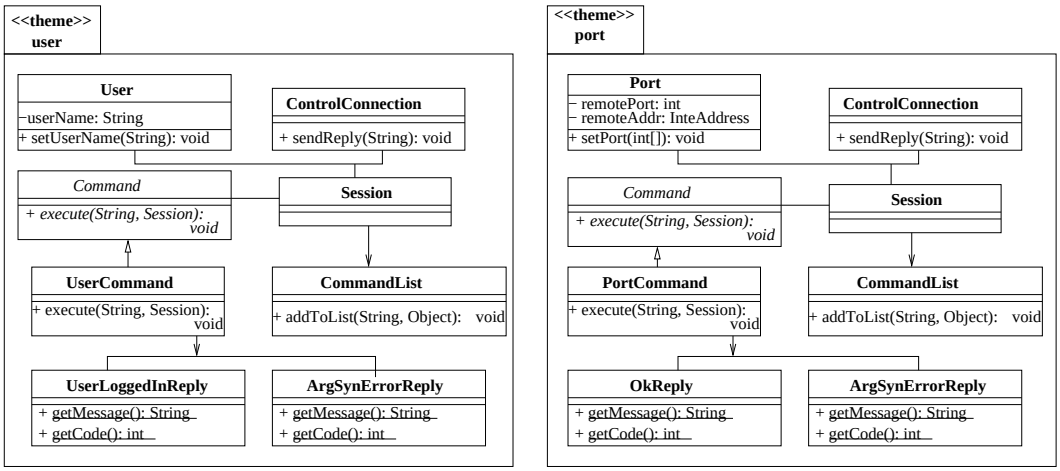

Figure B.7: `User` and `port` themes.

the shared concerns; it was not clear whether each theme should implement the shared behaviour, or would only refer to it, considering the shared behaviour is implemented by some theme. We attempted to assign a shared system behaviour to one design theme (to encapsulate), so that other themes only refer to it without implementing the complete details. Deciding up-front about implementation of the shared behaviour, might help addressing overlap of work.

The individual themes being developed together, a uniform communication protocol could be ensured for individual themes to interact among one another. As a result, no differences in interactions among the themes were observed, as we attempted to integrate the themes into a complete system. Moreover, no conflicts or difference among the views of individual themes could arise, since they were consid-

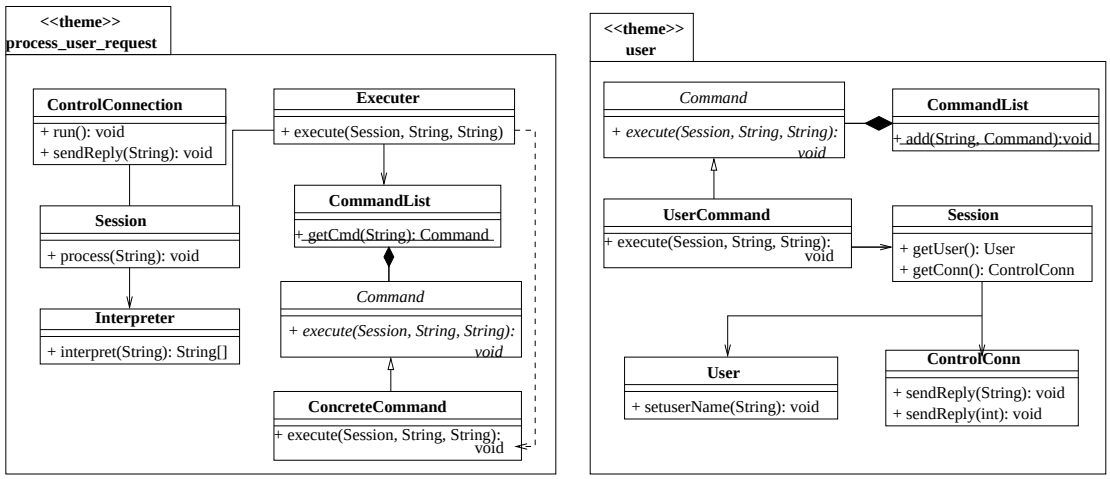

Figure B.8: The themes share a common design architecture.

ered to be developed together. The Theme/UML composition rules of merge and override integrations sufficed for a successful integration of the resulting themes into a complete functioning system (version 1).

B.4 Communications in the late aspect model (Path 2.2)

In investigating application of the late aspect model of Theme, we observed that by providing a common design architecture for individual themes to conform to, could resolve communications among the themes in the integrated system.

Figure B.8 shows two design themes, `process_user_request` and `user`, developed considering individual themes share a pre-defined design architecture (along Path 2.2), as discussed in Chapter 4. In `process_user_request` theme, the

`Session.process(String)` operation, after interpreting a user-request (passed as argument), invokes `Executer.execute(..)` operation. The following piece of Java code represents the `execute(..)` operation.

```
public void execute(Session session, String command, String argument){
    FlyweightFacotry.getFlyweight(command).execute(session, command, argument);
}
```

`User` theme (in Figure B.8) expects to be communicated with by a call to the `UserCommand.execute(..)` operation; it stores the user-name passed as an argument and sends a corresponding reply-message. The theme also updates the `CommandList` class, as defined by the pre-planned design.

Composition of the two themes is expected to provide a correct communication, whenever a user sends a request for the FTP-command, *user*. Let us consider communications between the two themes (in Figure B.8) upon their composition with Theme/UML. Implementation of `CommandList` class after composition would be as follows (represented as a Java code):

```
public class CommandList {
    protected static Hashtable hash = new Hashtable();
    public static void add(String cmdName, Command cmd){
    hash.put("user", new UserCommand());
    }
    public static Command getCmd(String cmd) {
     return (Command)hash.get(cmd.toLowerCase() );
    }
}
```

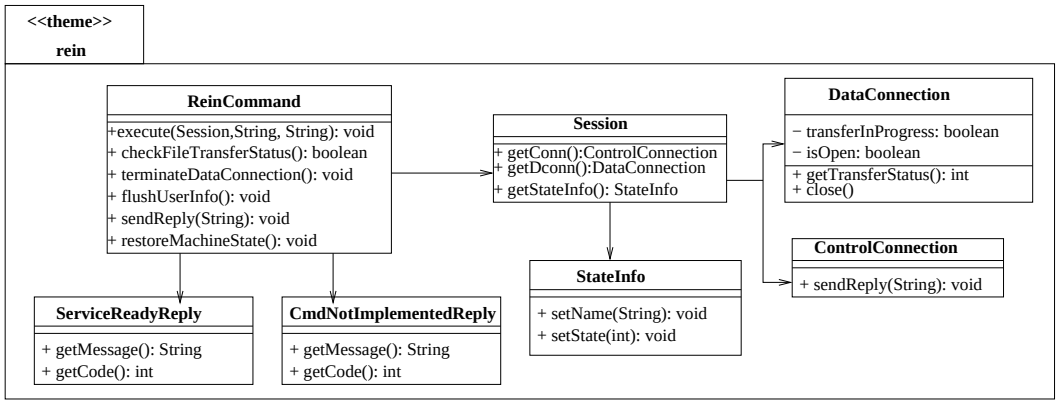

Figure B.9: `Rein` theme captures details for the command *rein*.

For a user-request for the FTP-command *user*, the `Executer.execute(..)` method would invoke the `UserCommand.execute(..)` operation, providing a correct communication between the themes. Similarly, communications for other themes could be possible following this common design architecture. The pre-defined design architecture sufficed to address the communications among individual themes.

B.5 Evolution step-3 with the late aspect model

In this section, we show a design for `rein` theme, which encapsulates details for the FTP-command, *rein* (re-initialize user), introduced in the third evolution step. The feature requires to check for any file transfer in progress. If so, the server would wait for the transfer to complete, close the data connection, flush the user information

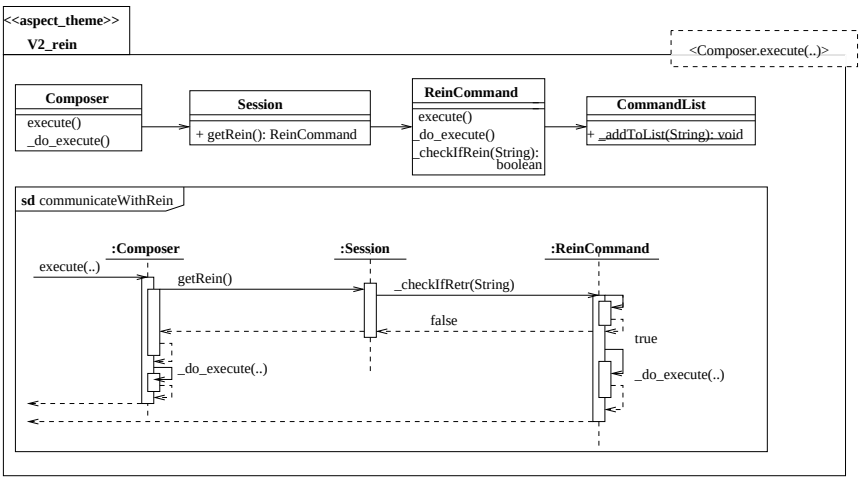

Figure B.10: The aspect addresses communication between `rein` and the existing system.

stored, and restore the default state, which corresponds to the initial state right after the establishment of a connection. Figure B.9 shows a design for the theme. A call to `ReinCommand.execute(..)` operation would trigger the encapsulating behaviour.

In integrating the theme with the existing system, we were required to address the communication between the system (Version 2) and `rein` theme. We added an aspect theme to address the crosscutting communication protocol; Figure B.10 shows a design for the aspect theme, that addresses communication between the new theme and the existing system, constructed along Path 2.1. A call to `Composer.execute(..)` operation triggers the crosscutting behaviour for the aspect theme; for the FTP-command, *rein*, `ReinCommand.execute(..)` operation is triggered and the triggering operation returns to base flow of control, otherwise the triggering operation is executed. Addition of the aspect theme could address the communication with the new theme, resulting in a complete Version 3.