Modeling variability and integration of architectural patterns

Kamal, Ahmad Waqas

RIJKSUNIVERSITEIT GRONINGEN

Modeling Variability and Integration of Architectural Patterns

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. E. Sterken,
in het openbaar te verdedigen op
vrijdag 27 januari 2012
om 12.45 uur

door

Ahmad Waqas Kamal
geboren op 24 december 1979
te Lahore, Pakistan

Promotor	:	Prof. dr. P. Avgeriou
Beoordelingscommissie	:	Prof. dr. K. Koskimies Prof. dr. N. Guelfi Prof. dr. M. Stall

Samenvatting

Architectuur patronen worden gedocumenteerd als oplossingen voor terugkerende ontwerp problemen die zich voordoen in specifieke ontwerp situaties. Software architecten maken gebruik van architectuur patronen om een bekende ontwerp problemen op te lossen. Architecturale patronen hebben een invloed op zowel de structuur als het gedrag van een systeem op architectuur ontwerpniveau. De structuur beschrijft de statische aspecten van een architectuur terwijl het gedrag richt zich op de run-time aspecten van de oplossing. Software architecten maken gebruik van bestaande model talen om patronen te modelleren tijdens de fase van de software architectuur ontwerp.

Architectuur patronen kunnen worden gemodelleerd in verschillende vormen om een ontwerp probleem op te lossen. Elk van deze vormen wordt een patroon variant. Systematische modellering van architectuur patronen is een uitdagende taak vooral als gevolg van de inherente variabiliteit patroon en omdat de architectonische abstracties van het modelleren van talen niet overeenkomen met patroon elementen. Verder worden architectuur patronen zelden gesoleerd toegepast op een software-architectuur. Vaak zijn meerdere patronen gentegreerd om een ontwerp probleem grondig aan te pakken. Echter, vanwege de abstracte aard van de huidige patronen kan de integratie van twee architecturale patronen verschillende vormen aannemen. Het is een uitdagende taak voor software architecten om architectuur patronen effectief te integreren.

Dit proefschrift draagt bij aan het oplossen van de twee eerder genoemde problemen door het ontwerpen van een aanpak voor het modelleren van patroon variabiliteit en patroon integratie dat software-architecten helpen bij het effectief toepassen van patronen in software architecturen. We proberen het patroon variabiliteit modellen probleem op te lossen door de oplossende deelnemers van patronen te categoriseren. Meer specifiek identificeren we variabele deelnemers die leiden tot specialisaties binnen de individuele patroon varianten en ontdekken terugkerende architecturale abstracties binnen enkele architectuur patronen genaamd architectuur primitieven. We laten zien dat het gebruik van de architectonische primitieven en pattern specifieke architecturale elementen een effectieve combinatie biedt voor het modelleren van patronen en patroon varianten. Het patroon integratie probleem wordt opgelost door het ontdekken van een relatie set van de deelnemers die de effectieve integratie van architecturale patronen aangeven. Onze bevindingen worden gevalideerd door middel van voorbeelden en gecontroleerde experimenten waaruit blijkt dat de voorgestelde primitieven de patroon variant-specifieke elementen, en hun relaties, ontwerpers ondersteuning bied in het modelleren van patronen.

Abstract

Architectural patterns are documented as solutions to recurring design problems that arise in specific design situations. Software architects use architectural patterns to solve known design problems. Architectural patterns have an impact on both the structure and the behavior of a system at the architecture design level. The structure describes the static aspects of an architecture while the behavior addresses the run-time aspects of the solution. Software architects use existing modeling languages to model patterns during the phase of software architecture design.

Architectural patterns can be modeled in several different forms to address a design problem at hand. Each such form is called a pattern variant. Systematic modeling of architectural patterns is a challenging task mostly because of the inherent pattern variability and because the architectural abstractions of modeling languages do not match pattern elements. Further, architectural patterns are seldom applied in isolation to design a software architecture. Often several patterns are integrated to fully address a design problem at hand. However, due to the abstract nature of current pattern relationship approaches and because the integration of any two architectural patterns can take several forms, it is a challenging task for software architects to effectively integrate architectural patterns.

This thesis contributes to address the two aforementioned problems by devising approaches for modeling pattern variability and pattern integration that assist software architects in effectively applying patterns to software architectures. We attempt to solve the pattern variability modeling problem by categorizing the solution participants of patterns. More precisely, we identify variable participants that lead to specializations within individual pattern variants and discover recurring architectural abstractions within several architectural patterns called architectural primitives. We demonstrate that the use of the architectural primitives and pattern-specific architectural elements in combination offers an effective way to model patterns and pattern variants. The pattern integration issue is addressed by discovering and defining a set of pattern participants relationships that serve the purpose of effectively integrating architectural patterns. Our findings are validated through examples, case studies, and controlled experiments which provide evidence that the proposed primitives, pattern variant-specific elements, and relationships support designers in modeling patterns.

Contents

Acknowledgment

Several people assisted and inspired me during my PhD work. I am obliged to following people for their support.

First of all, thanks to my promoter Paris Avgeriou. He allocated a good zeal of his precious time to supervise me. We regularly met every week during the first four years of my PhD work. Later, he always allocated time for meetings whenever I requested him. Paris guided me to effectively spend my time at work. He also encouraged me to participate in extracurricular activities. We had numerous technical meetings and I always found him helpful to my research ideas. I am sure that I will keep in touch with him in future as well.

Uwe Zdun also helped me a lot in my research work. It was a good experience to work with him and receiving feedback to my work via e-mails. I am also thankful to Nick Kertley, Robert Vrooland, Samuel Esposito and Johan Drenthen for their participation in implementing an open source tool as part of my research work. In addition, several current and former colleagues helped me make my stay wonderful and knowledgeable during my stay at University of Groningen; in particular, Neil Harrison, Anton Jansen, Peng Liang, Dan Tofan, Trosky Callo, Matthias Galster , Eirini Kaldeli, Ehsan Warriach, Viktoriya Degeler, Pavel Bulanov, Andrea Pago, George Azzopardi, Charmaine Borg, Giannis Giotis, Kerstin Bunte, Mahir Can Doganay, and Ellie El-khouri.

I would also like to thank the secretaries, Ineke Schelhaus, Esmee Elshof, and Desiree Hansen, for helping me solving university administration related problems to attend conferences and workshops. Thank you for helping me out.

Finally to wrap-up this acknowledgement, I would like to thank my parents and my wife for their love and support. My father encouraged me to achieve this milestone and my mother always prayed for my success. Thanks to my wife Komal for her love and support. She always co-operated with me during my PhD work and especially during the phase of thesis writing.

Chapter 1

Introduction

1.1 Software architecture

Software architecture is a fast growing discipline in the area of software engineering. The software architecture field is focused on the idea of reducing system development complexity through high-level design and separation of concerns [1]. A software architecture of a system is used as a guideline during the development process, facilitates communication between stakeholders, describes high-level design and it is also used to negotiate system requirements [2]. The term software architecture was first introduced in the late sixties and was related to decomposing a system at a high level of design abstraction [1]. However, it was not until the late nineties that software architecture became an essential part of developing software. Designing an overall structure of the system becomes essentially more important as the design problem goes above computations, functionality and algorithms [1]. Design problems at architecture level include communication protocols, synchronization, data access, assigning functionality to architectural elements, and selection among design alternatives [3]. Designing the architecture of a system forces the architect(s) to consider the key design aspects early and across the whole system. Because the software architecture is the link between the system requirements and development, this design activity starts after the domain and requirements analysis, and leads to detailed design, coding, integration, and testing [2].

The architecture of every system is unique due to the nature of the requirements it addresses such as the quality requirements exhibited by a system e.g. performance, maintainability, scalability, security etc. Designing software architecture still largely depends on an architect's intellectual and technical skills. Earlier problems of complexity were addressed by software architects by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. However, initial efforts to effectively design software architectures were imprecise and disorganized, often architectures were designed using a set of box-and-line diagrams [3]. During the nineties a more focused effort was put in place by experts to define and document the fundamental aspects of the field [1]. In late nineties software architecture design community started documenting known solutions to architectural problems in the form of patterns. Initial sets of design patterns and styles were documented during that time as further discussed in next section.

1.2 Patterns in software architecture

Experienced software engineers make common use of architectural principles when designing complex software. Many of the principles are in the form of documented patterns that have come forth informally over time. Architectural patterns have emerged as a paradigm to effectively design software architectures. Architectural patterns are used as building blocks for designing large scale software systems. They can be used to specify particular aspects of software architecture e.g. distributed components, remote procedure calls, layering etc. Every

pattern provides a predefined set of components and relationships between them. Capturing the commonality that exists in different system designs allows developers to take advantage of knowledge they already posses, applying known techniques to solve design problems [4].

Architectural patterns document successful experiences of designers for solving recurring design problems that arise in a system context. They specify guidelines for designing the structural and behavioral aspects of a system. When designing a software architecture, the selection of architectural patterns is often one of the fundamental design decisions [5]. An architectural pattern provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [5]. However, patterns are not invented or created, rather they are discovered and documented by experienced architects to reuse a design solution. Other architects, familiar with the documented patterns, can apply them directly to solve a design problem at hand. For instance, to design a data processing application, the Pipes and Filters architectural pattern can be applied to decouple different data processing steps to support incremental data processing. Each filter can represent a self-contained data processing unit while pipes pass data along the filter chain.

Although patterns are now understood as solutions to known design problems in the field of software development, they originated in the physical world of designing architectures for buildings and cities. The architect Christopher Alexander identified the concept of patterns for capturing architectural decisions and arrangements [4]. He documented the guidelines on which many of today's pattern approaches are built. Alexander described over two hundred and fifty patterns at different scale and abstraction, mostly for structuring towns, regions and houses. He also defined the fundamental Context-Problem-Solution structure of patterns [6]. The use of the term 'pattern' was first introduced by Christopher Alexander who has written several books on the topic.

The pioneers of patterns theory for software development are Ward Kunningam and Kent Beck [6]. Kunningam and Beck's first five patterns deal with the design of user interfaces. The patterns mined by them established the beginning of patterns use in software development field [6]. Design patterns became well-known in computer science after Gamma et al. [7] published the book titled "Design Patterns: Elements of Reusable Object-Oriented Software" in 1994. Other books that have helped popularize patterns are Pattern-Oriented Software Architecture books series [5][8][9][4] [10] [11] [6] and proceedings from the Pattern Languages of Program conferences [12]. At present, the software community is using patterns largely for designing software architecture, and most recently software development processes and organizations [6].

In the early days in mid 90s, the focus was on object oriented design patterns. The object oriented design patterns book [7] presents the most widely-known patterns of this type. However, the scope of these patterns have a limited influence on software architecture. This gap was first filled in 1996 with the publication of the Pattern-Oriented Software Architecture book [5] which was the first to present patterns[1] at software architecture level. Since then patterns have spread into many other areas of software development ranging from concurrent networked systems [8], security patterns [6], server components [10], human-computer interaction [13], and resource management [9].

Many other software experts followed the work done by Gamma et al. and Buschmann et al. producing an almost endless list of publication on patterns.

[1] The terms pattern(s), frequently used in this thesis, refers to architectural pattern(s)

1.3 Problem statement

Architectural patterns offer generic solutions to known design problems. The solutions specified by architectural patterns can be modeled in many different forms and often several patterns are used to design an architecture. This means an architect has to specialize a pattern's solution to model a pattern variant and integrate it with related patterns to fully address a design problem. However, modeling pattern variability and integrating patterns are two of the main challenges for effectively modeling patterns as elaborated in the following two subsections.

1.3.1 Variability in modeling patterns

Architects select architectural patterns to solve specific design problems. Patterns provide generic solutions to known design problems and leave blank spaces to be filled in by an architect. That means that the architect has to specialize the solution offered by patterns, often using a modeling language. It is therefore not easy to provide prefabricated design solutions that can be applied 'as is' to an architecture. The generic solution of patterns can be specialized in several different forms to meet the design requirements at hand. Each such specialization can lead to a unique variant of a pattern. In essence, each architectural pattern entails numerous variations in which it can be applied to design a software architecture which has its pros and cons on the resulting software architecture. For example, if the layered architectural pattern has been chosen to design an architecture, then variants of this pattern may be the strict layered pattern (where each layer is only allowed to call its immediate subordinate layer) and the relaxed layered pattern (where each layer can invoke all lower-level layers, rather than just the layer immediately below it). Using the strict layered pattern increases flexibility, but generally decreases performance of a system. Using the relaxed layered pattern improves performance, but influences maintainability negatively [14].

The aforementioned challenge of variability in patterns solution is to-date a major challenge for effectively expressing different variants of patterns. An architect has to put extensive design effort to express a pattern variant's solution using a modeling language. Often, the participants of architectural patterns and pattern variants do not match the architectural elements present in modeling languages. For instance, the Pipe participant of the Pipes and Filters pattern do not match the Connector element of UML [15]. While patterns offer generic solutions that can be used in several different system contexts, current modeling approaches provide limited support to effectively grasp the whole solution space covered by architectural patterns and their variants.

1.3.2 Integrating architectural patterns

Architectural patterns are often applied in combination with related patterns to a software architecture. However, possible pattern combinations are currently not explicitly addressed by existing pattern relationship approaches [5] and modeling languages [16] mainly because of the following two reasons:

- Existing pattern languages document associations between patterns at a generic level but do not go into details concerning the relationships between the pattern participants. Pattern languages do not elaborate on how the participants of the patterns collaborate. The relationships among architectural patterns participants are important to effectively address the design requirements. The details of such relationships between participants

concern for example how participants of related patterns overlap, interact, or override other participants in the resulting software architecture. While the relationships between patterns are visible in current pattern languages, they provide little support to an architect in integrating patterns.

- The integration of two selected architectural patterns does not always result in one particular solution but leads to several possible design solutions depending on the system context at hand. In other words, pattern-to-pattern relationships are not always fixed but may entail a great deal of variability. For instance, to model interactive applications, the Model-View-Controller (MVC) [5] and Layers [5] patterns can be integrated in several different forms. In the 2-tier layered variant, the presentation layer consists of the View and Controller participants while the application logic layer owns the Model participant. However, in the 3-tier application architecture, the View may correspond to user interface layer, Controller correspond to business layer, and Model correspond to data logic layer. Systems generally use multiple patterns in their architecture that have several different possible combinations, this variability in pattern combinations is currently not explicitly addressed by existing pattern languages.

In this thesis, I address the problems discussed in the above two subsections by presenting pattern modeling techniques that can help software architects to model pattern variability as well as integrating architectural patterns.

1.4 Research questions

The work presented in this thesis attempts to answer the following, overall research question:

RQ: *How can architects effectively model pattern variability and integrate patterns during the phase of software architecture design?*

In order to address the research question **RQ**, we first need to understand the level of support that present modeling languages offer for modeling architectural patterns. To this end, we formulate the following research question:

RQ-1: *What support do the existing modeling languages offer for modeling architectural patterns?*

We analyze the support that few modeling languages provide in modeling a small set of architecture patterns. The results highlight the strengths and weaknesses of the modeling languages for expressing architectural patterns. We identify that existing modeling languages provide limited support for modeling the solution specified by architectural patterns. Some of the languages provide built-in solutions for modeling few patterns and few others offer extensibility support to model patterns. In particular, we notice that effectively modeling patterns using a modeling language requires extensive design effort mainly because of the several different forms of a pattern that can be used to design a software architecture. Each such form is called a pattern variant. Before we could address modeling several variants of patterns, we first start with working on specific variants of architectural patterns and formulate the following research question.

RQ-2: *How to effectively model the solution of a specific pattern variant?*

We propose recurring architectural abstractions, called architectural primitives, discovered in the solution of several patterns, for modeling pattern variants. The architectural primitives

discovered during this work are in addition to an existing set of primitives documented in [15]. We investigate the use of architectural primitives for modeling few pattern variants in structural and behavioral views. Using this approach we successfully modeled specific variants of patterns. We identified that often, in addition to the use of primitives, architects need to put extra design effort to express the missing aspects of a pattern's solution that are not addressed by primitives. This lead us to realize the need of an approach that can help architects effectively model any pattern variant. With an understanding of how a specific pattern variant can be expressed using architectural primitives, we are prepared to consider how to systematically model any variant of a pattern in a way that requires minimal design effort as discussed further in the following research question:

RQ-3: *How to effectively model the solution of any variant of a pattern?*

An approach is described that uses a set of architectural primitives and a vocabulary of pattern variant-specific architectural elements for the systematic modeling of any variant of a pattern. We propose to categorize the solution participants of architectural patterns. The concept of generic pattern participants is introduced which lead to several specialized pattern participants within individual pattern variants. To validate the applicability of the proposed approach, results from a controlled experiment are presented. Our work up-to this point has focused on modeling the variability of individual patterns. However, during real software architecture design, as discussed in subsection 1.3.2, patterns are seldom applied in isolation to design a software architecture and often several patterns are integrated to address design problems at hand. At this stage, we realized the need of an approach that can help software architects for expressing several patterns and pattern variants in combination, as is formulated by the following research question:

RQ-4: *How to effectively integrate architectural patterns and pattern variants within software architectures?*

We propose to address the pattern integration issue by discovering and defining a set of pattern participants relationships that serve the purpose of effectively integrating several patterns and pattern variants. A controlled experiment is performed to validate the effectiveness of pattern participants relationships for integrating architectural patterns.

1.5 Research methodology

1.5.1 Introduction

Unlike other fields of research, software engineering research does not have well-understood guidelines for research strategies [17]. A number of attempts to characterize software engineering research have made contribution to address the issue but they do not yet paint a comprehensive picture [18]. In 1980, Shaw [19] examined the relations of different engineering disciplines and laid out expectations for an engineering discipline in software. In 1985, Redwine and Riddle [20] proposed a model for the way software technology evolves from research idea to widespread practice. During the past few years, several research processes have been introduced in the field of software engineering [17]. Some researchers in the field of software engineering have developed research classification frameworks, like Shaw[19], and some others have detailed the guidelines to conduct experiments and report results [21]. Following we discuss the approach followed in this thesis for the design and validation of the proposed research

work.

1.5.2 Research methods

Depending on the kind of problem to solve and the context of the problem, Science or Engineering, different research methods are used [17] [19]. There is not any precise method to address software engineering research problems [18]. Shaw [19] documents several questions related to the selection of research methods, like:

- What do you want to achieve?
- Where does the data come from?
- What will you do with the data?

To answer such questions, Holz et al. [17] document 55 research methods and Glass et al. [22] summarize 19 research methods. Of these methods, the following three research methods are used in this thesis:

- Static analysis: This research method involves study and collection of data from published material [23]. Often used for evaluation purpose, an evaluation framework can be created that describe criteria on which the information fetched from published material is evaluated. This method is used in the following research question:
  - **RQ-1:** *What support the existing modeling languages offer for modeling architectural patterns?* We use the static analysis technique, more precisely conduct a literature review, by devising an evaluation criteria to analyze the support that few modeling languages offer for expressing architectural patterns.
- Developmental research: This method deals with developing and describing methodologies and approaches to support general system development, or development of specific types of systems or system components [24]. Developmental research is used in this thesis to answer the following research questions:
  - **RQ-2** *How to effectively model the solution of a specific pattern variant?* This research focused on the use of recurring architectural abstractions repeatedly found in the solution of different architectural patterns, called architectural primitives, to model pattern variants in structural and behavioral views of an architecture; see Chapter 3 and Chapter 4 for the use of primitives for modeling pattern variants.
  - **RQ-3** *How to effectively model the solution of any variant of a pattern?* This research developed a pattern variability modeling approach for modeling several variants of a pattern by categorizing the solution participants of architectural patterns, which is described in Chapter 5.
  - **RQ-4** *How to effectively integrate architectural patterns and pattern variants within software architectures?* This research proposed the relationships between the participants of architectural patterns as a mean to integrate architectural patterns for designing software architectures; see Chapter 6 for the use of pattern participants relationships in integrating patterns.
- Laboratory experiment: This method is an experiment in controlled settings, involving independent and dependent variables to test hypotheses. In this thesis, the aforementioned developmental research in **RQ-3** and **RQ-4** is further validated using laboratory experiment method as follows:

- **RQ-3:** *How to effectively model the solution of any variant of a pattern?* We organized a controlled experiment where one group was instructed to use the variability modeling approach while the other group was not provided such support. The resulting architectures from both groups were compared to analyze the performance of participants for modeling pattern variants.
- **RQ-4:** *How to effectively integrate architectural patterns and pattern variants within software architectures?* Similar to the aforementioned experiment, we organized another controlled experiment where one group was provided coarse grained associations between the participants of patterns and a control group without it, to compare the resulting architectures between the groups.

1.5.3 Research question types

Research questions can be about methods for developing software, about analyzing software, about the design, evaluation, and implementation of specific methods, or about generalizing over whole class of systems [17]. Shaw [18] has documented several types of research questions in the field of software engineering. In this thesis, three kinds of research questions are addressed:

- *Method for analysis:* This method is aimed at extracting information by analyzing the data to reach a conclusion or to make a comparison. For instance, how can I evaluate the quality and correctness of X? or how do i choose between X and Y? [18]. **RQ-1** is about a method for analysis where we want to evaluate the strengths and weaknesses of different modeling languages for expressing architectural patterns.
- *Method or mean of development:* The research question of this type deal with questions about automating software development, and improving ways to design them. For example questions like how can we automate doing X? or is that a better way to do X? [18].
  - **RQ-2** *How to effectively model the solution of a specific pattern variant?* This research question searches for the ways to effectively model pattern variants in structural and behavioral views.
  - **RQ-3** *How to effectively model the solution of any variant of a pattern?* This research question seeks for an approach to systemize the pattern variability modeling for designing software architecture.
  - **RQ-4** *How to effectively integrate architectural patterns and pattern variants within software architectures?* This research question searches for an effective way to integrate architectural patterns for designing software architectures.
- *Design, Evaluation, or analysis of a particular instance:* In addition to the aforementioned developmental research about **RQ-3** and **RQ-4**, the validation work for both of these questions involves this type of research; we conduct controlled experiments to evaluate a pattern variability modeling approach (**RQ-3**) and a pattern integration approach (**RQ-4**).

1.5.4 Research results

Shaw[18] reports several kinds of research results in software engineering. The result may be a specific procedure, technique for software development, or analysis [18]. We use the following types of research results in this thesis:

- Answer or judgment: The results of this type are specific analysis, evaluation, or comparison. The results for **RQ-1**, as documented in Chapter 2, are judgment about the support that modeling languages provide for modeling patterns.
- Procedure or technique: The result can be new or better way to do some task, design, implementation, measurement, evaluation, selection from alternatives, operational techniques for implementation, representation, management, or analysis. The results of research questions **RQ-2**, **RQ-3** and **RQ-4** are of this type. Examples of this research technique in this thesis are: a) the use of architectural primitives for modeling pattern variants is a research technique presented in Chapter 3 and Chapter 4, b) the categorization of pattern's solution participants for modeling pattern variability is a research technique presented in Chapter 5, and c) the use of pattern participants relationships for integrating patterns is another research technique described in Chapter 6.
- Qualitative or descriptive model: Such a model presents structure or taxonomy for a problem area in the form of well-grounded checklists or well-argued informal generalizations etc. Example of this result type are: a) Chapter 2 that presents an overview of the strengths and weaknesses of existing modeling languages for expressing patterns, b) Chapter 3 and Chapter 4 that document catalogs of architectural primitives in structural and behavioral views.
- Report: A report about interesting observations, judgments, and discovered rules of thumb as result of descriptive or quantitative analysis. In this thesis, Chapter 5 and Chapter 6 present this kind of research results. In these chapters we analyze the qualitative and quantitative data and make judgments about the performance of participants in the control and treatment groups; see subsection and subsection.
- Notation or tool: This type of research result include formal or graphical language or tool support. The Primus tool (see Appendix B) offers tool support for modeling patterns and pattern variants. More examples of this type of research results are UML models in Chapter 3 and Chapter 5.

1.5.5 Validation of results

Validation of a proposed approach helps researchers and practitioners to use it for their own work or extend it for other purposes. There are many different validation techniques in software engineering. This section details the validation techniques used in this thesis as detailed below:

Example:

Examples on how the proposed research ideas can work are documented. Two types of examples are presented in this thesis:

- **Toy example:** Simplified examples to demonstrate the use of approaches presented in this thesis. This validation technique is mostly used in this thesis to demonstrate the modeling of few selected architectural patterns and pattern variants. The complexity of the examples range from simple examples demonstrating the modeling of individual patterns to complex examples for designing parts of real software architecture. The validation technique is used in this thesis in: a) three architectural patterns are modeled as an example in Chapter 3, b) In Chapter 4, three architectural patterns are modeled using UML's interaction diagrams, c) In Chapter 5, two variants of a pattern are modeled as examples

to demonstrate the use of a pattern variability modeling approach and, d) In Chapter 6, small examples for each relationship discovered between the participants of architectural patterns are presented.

- **Slice of life:** Case studies are a powerful and flexible empirical method [25]. They are used primarily for exploratory investigations, both prospectively and retrospectively, that attempt to understand and explain phenomenon or construct a theory. They are generally observational or descriptive in nature. This validation technique is used in this thesis in Chapter5 and Chapter 6. In Chapter 5 we present a case study for designing an Image Processing System and in Chapter 6 a small part of the architecture of a Warehouse Management System is designed.

Analysis:

The following type of analysis is used in this work:

- **Controlled experiment:** Controlled experiments offer several specific benefits. They allow us to conduct well-defined, focused studies, with the potential for statistically significant results. They allow us to focus on specific variables, measures, and the relationships between them and help us formulate hypotheses by forcing us to clearly state the question being studied. Such studies usually result in well-defined dependent and independent variables and hypotheses. There is an increasing understanding in the software engineering community that empirical studies are needed to develop or improve processes, methods, and tools for software development and maintenance [26]. In this thesis, in Chapter 5, we conduct a controlled experiment to test the use of variant-specific solution participants and primitives in combination for modeling pattern variants. In Chapter 6, we conduct another controlled experiment to examine the use of pattern participants relationships for integrating patterns. In both of these chapters, the experiments are performed in accordance to the guidelines reported in [27].

Evaluation:

The criteria is documented against which the research results are presented. The following type of evaluation is used:

- **Descriptive model:** The results describe the phenomena of interest. Chapter 2 presents the application of this technique where we present an evaluation framework and apply it to different modeling languages. The results in this chapter provide a descriptive comparison of modeling languages for their support to model patterns.

1.6 Thesis outline

The research work presented in this thesis is based on scientific articles that are already published and one article that is submitted for review. The articles that have overlapping information are slightly modified and merged in respective chapters to make the style, structure and terminology consistent in the thesis. A brief summary of the chapters is as follows:

Research Question	Research Question Type	Research Result	Validation Technique	Chapters
RQ-1	method for analysis	answer or judgement	- evaluation	2
RQ-2	method or mean of development	- procedure or technique - qualitative or descriptive model	- answer or judgement - toy example - slice of life	3,4
RQ-3	design, evaluation, or analysis of a particular instance	- procedure or technique - qualitative or descriptive model - notation or tool	- answer or judgement - slice of life - controlled experiment	5
RQ-4	design, evaluation, or analysis of a particular instance	- procedure or technique - qualitative or descriptive model	- toy example - controlled experiment	6

Table 1.1: *Overview of the classification of research questions*

An evaluation of ADLs on modeling patterns for software architecture

Chapter 2, published at the 3rd International Workshop on Rapid Integration of Software Engineering techniques [28], is a survey of the pattern modeling support offered by existing modeling languages and tools. Specifically, we attempt to evaluate the existing Architecture Description Languages (ADLs) for modeling architectural patterns. We establish a comparison framework that is composed of features needed in ADLs for effectively modeling architectural patterns. Using this framework, we evaluate the most popular or commonly used ADLs, with respect to four of the most significant architectural patterns. The results highlight the strengths and weaknesses of ADLs for modeling patterns. The evaluation answers research question **RQ-1** and provides deeper understanding for the support of ADLs for modeling patterns. The contribution of the chapter helped us understand the state of art for modeling patterns using modeling languages. The fact that the existing languages provide weak support for modeling patterns lead us to devise approaches for effectively modeling the solution specified by patterns as further discussed in subsequent paragraphs.

Chapter: 2, **Research Question:** RQ-1

Modeling architectural patterns variants (Chapter 3) *and*
Modeling Architectural Patterns Behavior Using Architectural Primitives (Chapter 4)

Chapter 3, published at the European Pattern Languages of Programming conference [29], and Chapter 4, published at the 2nd European Conference on Software Architecture [30], describe the use of architectural primitives for effectively modeling architectural patterns in structural and behavioral views respectively. The architectural primitives documented in these chapters were found among a number of architectural patterns and served as the basic building blocks for modeling specific pattern variants. We use a set of architectural primitives and a vocabulary of pattern-specific architectural elements for modeling pattern variants. To define these concepts, we extend the UML metamodel for each discovered architectural primitive us-

ing UML profiles and define UML stereotypes as extension to UML elements for defining a pattern's solution participants. Chapter 3 presents three pattern variants in the Component and Connector view (i.e. structural view) and Chapter 4 demonstrate the modeling of another three pattern variants using UML's interaction diagrams (i.e. behavioral view). The results demonstrate that the use of the primitives along with the stereotyping scheme is capable of handling some of the challenges for effectively modeling specific pattern variants in different architectural views.

Chapters: 3 and 4, **Research Question:** RQ-2

Modeling the variability of architectural patterns

Chapter 5, submitted for publication in Software Practice and Experience journal, is an extended version of our work [31], published at Software Engineering track of the 25th Symposium On Applied Computing. In this chapter, we propose to categorize the solution participants of architectural patterns for effectively modeling pattern variability. The concept of default pattern participants as variation points is introduced which lead to several specialized pattern participants within individual pattern variants. More precisely, we identify variable pattern participants that lead to specializations within individual pattern variants. These specialized participants are then used in combination with architectural primitives for modeling any pattern variant. With examples and a case study, we demonstrate the successful applicability of this approach for designing software architectures. Further, we explored the benefit of using such an approach by performing a controlled experiment in which architects were divided in two groups: control and treatment. All participants in both teams were asked to individually design a medium scale software architecture. In addition, the treatment group was restricted to use primitives and pattern participants for modeling pattern variants. The majority of participants in the treatment group were better able to effectively model pattern variants by producing better quality architecture as compared to the control group. This validates our work that the primitives and pattern participants information is useful to software architects for modeling any variant of a pattern.

Chapter: 5, **Research Question:** RQ-3

The Use of Pattern Participants Relationships for Integrating Patterns

Chapter 6, published in special issue on pattern languages of the Journal of Software Practice and Experience [32], is an extended version of our work [33], published at the 4th European Conference on Software Architecture. In this chapter, we propose to address the pattern integration issue by discovering and defining a set of pattern participants relationships that serve the purpose of effectively integrating several patterns. The relationships presented in this chapter are based on the study of software architectures from 32 industrial software systems, pattern integration examples documented in the literature, and patterns presented in workshops and conferences. The patterns integrated within real software architectures are analyzed to discover the relationships between the participants of related architectural patterns. The underlying idea behind our approach is that various architectural patterns can be effectively integrated using a set of 'pattern participants relationships'. Our findings are validated through a controlled experiment. Two groups of software architects were asked to design a software architecture by integrating architecture patterns. One group was given pattern participants relationships information, the other group was not provided such information. The participants in both groups were asked to individually design a software architecture. From the software architectures gathered after the experiment, which were analyzed by external reviewers, the group with the pattern participants relationships information managed to more effectively integrate architectural patterns by producing better quality software architecture as compared to

the other group. The results from the experiment provided us with significant evidence that the proposed relationships support software architects in integrating patterns.

Chapter: 6, **Research Question:** RQ-4

The final chapter of this thesis Chapter 7 documents the research questions and answers, describes the contribution, and concludes this study. In this chapter, we describe some open questions that can motivate further work in the field. We argue that more research can be carried that will improve the state of the art for modeling architectural patterns for designing software architecture.

Published as: A. W. Kamal and P. Avgeriou – *"An Evaluation of ADLs on Modeling Patterns for Software Architecture,"* 3rd International Workshop on Rapid Integration of Software Engineering Techniques (RISE), November 26-27, 2007, Luxembourg.

Chapter 2

An Evaluation of ADLs on Modeling Patterns for Software Architecture

Abstract

Architecture patterns provide solutions to recurring design problems at the architecture level. In order to model patterns during software architecture design, one may use a number of existing Architecture Description Languages (ADLs), including the UML, a generic language but also a de facto industry standard. Unfortunately, there is little explicit support offered by such languages to model architecture patterns, mostly due to the inherent variability that patterns entail. In this chapter, we analyze the support that few selected languages offer in modeling a limited set of architecture patterns with respect to four specific criteria: syntax, visualization, variability, and extensibility. The results highlight the strengths and weaknesses of the selected ADLs for modeling architecture patterns in software design.

2.1 Introduction

Architecture patterns [34] [5] entail solutions to recurring architecture design problems and thus provide a systematic way to architecture design. They offer re-use of valuable architectural knowledge, understanding, and communication of software architecture and support for quality attributes [5]. Architecture patterns are usually described and therefore modeled as configurations of components and connectors [35]. The components comprise the major subsystems of a software system and they are linked through connectors, which facilitate flow of data and define rules for communication among components. Examples of connectors are shared variable accesses, table entries, buffers, procedure calls, network protocols, etc. [36]. Connectors play a major role in modeling patterns for software architecture design.

In current software engineering practice, architecture patterns have become an integral part of architecture design, and often modeled with the use of Architecture Description Languages (ADLs): specialized languages for explicit modeling and analysis of software architecture [16]. UML is also used in practice for modeling software architecture, and we shall include it in the general category of ADLs, even though it is not strictly speaking an ADL. These languages are required to not only model general architecture constructs, but also pattern-specific syntax and semantics. Indeed, few ADLs, like Aesop [35], UniCon [37], and ACME [38] provide some inherent support for modeling specific concepts of architecture patterns. However, ADLs lack explicit support for modeling patterns, and are too limited in the abstractions they provide to model the rich concepts found in patterns [15] [35] [38].

In this chapter, we attempt to evaluate the strengths and weaknesses of existing ADLs for modeling architecture patterns. We establish a comparison framework that is composed of features needed in ADLs for effectively modeling architecture patterns. Using this framework, we

evaluate the most popular or commonly used ADLs, with respect to four of the most significant architecture patterns. The comparison framework consists of the following criteria:

- Syntax - expressing pattern elements, topology, constraints and configuration of components and connectors
- Visualization - graphical representation for modeling patterns
- Variability - the ability to express not only individual solutions but the entire space of solution variants
- Extensibility - capability to model new patterns

Our purpose is to evaluate the capabilities of ADLs with respect to modeling architecture patterns. It is not a scorecard to compare one ADL against other ADLs; rather it facilitates architects to select ADLs that best meet their needs to model architecture patterns. The focus of this chapter is on domain independent languages. For the evaluation, we have selected six languages: UML, ACME, Wright, Aesop, UniCon and xADL. To make the aforementioned criteria workable, we use four different architecture patterns, namely Layers, Pipe-Filter, Blackboard, and Client-Server. The selection of these ADLs and patterns is not exhaustive but serves the purpose for a first evaluation of ADLs w.r.t. modeling patterns.

The remainder of this chapter is organized as follows. In section 2.2, we introduce the theoretical background of patterns and current state of the practice in modeling patterns. Section 2.3 explains the comparison framework, while the evaluation of the languages is presented in section 2.4. Section 2.5 contains related work and Section 2.6 wraps up with conclusions of this work.

2.2 Theoretical Background and State of the Practice

2.2.1 Architecture Patterns

During the last decade, there has been a considerable effort for the systematic re-use of architecture patterns as solutions to recurring problems at the architecture design level [9] [39] [40]. Numerous architecture patterns are in use and this list is growing continuously [9] [10]. Some of the research activities in the pattern community for the past few years have been: discovery of new patterns [5] [7], combined use of patterns as pattern languages [41] [12], and using patterns in software architecture design [35] [38] [42] [37].

Among a number of software patterns that exist in the literature, architectural patterns, and design patterns [43] are the most widely known and used. It is difficult to draw a clear boundary between both types of these patterns, because it depends on the way these patterns are perceived and used by software architects. The work in POSA [5] lists some traditional architectural patterns, while work in GOF [7] lists 23 specific solutions to design problems. GOF is more concerned about object-oriented issues of the system design, while the work in POSA is more concerned about architecture issues, i.e. high-level components and connectors. In this chapter we focus on the latter.

Another terminological difference that often causes confusion is that between architecture patterns [5] and architecture styles [44]. These two terms come from two different schools of thoughts. Their commonality lies in that both patterns and styles specify a certain structure, e.g. the 'Layers' pattern/style decomposes system into groups of components at a particular level of abstraction and enforces communication rules. Their differences are the following:

- In the architecture patterns perspective, patterns specify a problem-solution pair, where problem arises in a specific context and a proven general solution addresses that problem. A context depicts one or more situations where a problem addressed by the pattern may occur. Moreover, the patterns capture common successful practice and at the same time, the solution of the pattern must be non-obvious [41].

- In the architecture styles perspective, styles are defined as a set of rules that identify the types of components and connectors that may be used to compose a system [40]. Architecture styles are more focused on documenting solutions in the solution domain [40]. The problem and the rationale behind a specific solution receive little attention [41].

These two schools of thought have more or less converged admitting that they are indeed referring to the same concepts [5] [45]. We concur with this trend. For the sake of simplicity, we shall use only the term 'architecture pattern' in this chapter.

2.2.2 Modeling Architecture Patterns

Many researchers have focused on using the inherent as well as the extensible support of ADLs to model architecture patterns [15] [35] [46] [47]. Many of these ADLs focus on the use of components and connectors as architecture building blocks [48] and some provide built-in support to model patterns in software design. For instance, ACME supports templates that can be used as recurring patterns, Aesop allows pattern-specific use of vocabulary, and UniCon provides syntax and graphical icons support for a limited set of patterns. While describing architectures using ADLs, the architects mostly focus on the components as a central locus of computation for decomposing system functionality and use connectors as communication links between components. Furthermore, in an effort to bring ADLs closer to each other, some researchers are working with integrative approaches among ADLs [38], and among ADLs and UML [46]. However, these practices are still in an experimental phase, and there is yet no proven approach to model architecture patterns effectively. Unfortunately, the current practice of modeling architecture patterns is still un-systematic and ad-hoc.

2.3 Evaluation Framework

The framework elements defined in this section are used to assess the support offered by ADLs to model patterns. Four elements make up this evaluation framework: syntax, visualization, variability, and extensibility.

- Syntax. We define syntax as pattern-specific elements and rules that govern the modeling of architecture patterns e.g. grouping in Layers, communication links, topology in Client-Server, etc.

- Visualization. Graphical support for modeling patterns can be helpful in visual composition of pattern elements and graphical icons to represent pattern elements.

- Variability. Architecture patterns are characterized by an inherent variability, as they not only provide a unique solution to a problem but an entire solution space. The chosen variants in the different variation points affect the design, and quality attributes of the system. For instance, bypassing Layers in the layered pattern can affect maintainability. An important aspect of our work is to see how the variability in modeling patterns is addressed by ADLs.

- Extensibility. Discovery of new patterns and inclusion in the existing list of patterns requires extensibility of the ADLs. It is possible that the introduction of new patterns may entail new modeling elements, may introduce new constraints and rules etc. Therefore ADLs need to be extended to be able to model newly discovered patterns

2.4 Modeling Patterns in ADLs and UML

To evaluate the suitability of ADLs for modeling architecture patterns, we have selected UML [49] [46] and five ADLs for evaluation: ACME [38], Wright [42], Aesop [35], UniCon [37], and xADL [50]. Each of these languages provide unique support for modeling certain concepts of architecture patterns. UML provides explicit extensibility support for expressing pattern elements. ACME is used as an ADL and as an interchange platform between different ADLs and provides templates for capturing common recurring solutions. Wright provides enriched communication protocols. Aesop has a generic vocabulary of extensible architecture elements for expressing patterns. UniCon supports abstractions for a limited set of traditional architecture patterns. Finally, xADL uses XML tags and schemas to provide extensibility support for expressing pattern elements. The selection of these ADLs is based on: a) their popularity for designing software architectures [51]; b) their maturity for modeling patterns [52]; c) their capability for describing software architectures [16]; and d) their generalized nature and independence of specific domains.

We have selected four patterns for the evaluation: Layers, Pipe-Filter, Blackboard, and Client-Server. We selected these patterns because they are the most commonly used in practice and they represent a number of different domains and concerns. Layers demands grouping of components, Pipe-Filter handles streams of data, Client-Server is frequently used in distributed systems, and Blackboard is for dynamic configurations. Although we limit ourselves to only four patterns, we emphasize that our study is not meant to be exhaustive. However, an analysis of the most representative patterns is able to highlight the pros and cons of the different ADLs in modeling patterns. In the following sub-sections, we use the evaluation criteria defined in the previous section to evaluate each of these languages.

2.4.1 Syntax

UML: UML is intended as a modeling language for many different areas and it lacks considerably in expressing pattern elements. For instance, pipes in a Pipe-Filter pattern do not match with UML connectors, since UML connectors cannot have an associated state or interface. Such shortcomings can be solved through the extension mechanism of UML, where its metamodel can be extended with profiles to express architecture patterns. In specific, a UML profile is comprised of tagged values, metaclasses, and stereotypes, that may be defined to support pattern-specific syntax [46]. UML also provides explicit support through the Object Constraint Language (OCL) to express constraints for modeling pattern elements. Thus, to fully express most of the architecture patterns and to define interactions among pattern elements, the UML metamodel elements must be extended.

ACME: In addition to the core ontology of seven basic architecture design elements, ACME provides a template mechanism, which can be used for abstracting common reusable architectural idioms and patterns [38]. ACME allows defining user specified constraints on architecture elements to model patterns. Violations of these constraints are automatically checked by ACME studio. To apply constraints on architecture elements, ACME allows two kinds of rules specification: invariants, violations of which are errors and heuristics, violations of which generate

warnings [47]. For instance, a heuristic rule can be defined to flag a warning message if a particular filter has more than two ports.

Wright: Rich connector support in Wright makes it a good option for patterns that heavily rely on connector and protocol specifications. Wright provides connector protocols as roles and glues, where glues can be used to define and constrain the behavior of interacting components. In Client-Server pattern, this role and glue specification for connectors allows constraining which clients can communicate to which server at architecture level. The glue specification can be used further to describe how clients and servers fit into the configuration [42] [53]. Furthermore, Wright constructs can be used in modeling dynamic systems [54], which is helpful in Blackboard and Client-Server patterns. For instance, clients can be aware of the state of a server at run-time to use the services more efficiently [10]. Wright provides support for constraints checking with the use of accompanying tools. For instance, with use of the FDR tool [55], Wright syntax can be checked for deadlocks in Client-Server, cyclic graphs in Pipe-Filter, and compatibility checking, etc. However, Wright demands conversion of its description into CSP [55] first so that the CSP compatible tools [55] like FDR can work for automated checking.

Aesop: Aesop is a system for developing pattern-specific architecture design environments for specifying pattern elements, topology, and constraints, etc. [35]. It provides a generic list of seven elements (i.e. components, connectors, ports, roles etc.), which can be customized to represent pattern-specific elements. This customization is based on the principal of subtyping: a pattern-specific vocabulary of design elements by providing subtypes of basic architecture elements [35]. For example, in the Pipe-Filter pattern, a port class can be sub-typed as Input and Output, and a role class can be sub-typed as Source and Sink. In addition, Aesop provides first class connector support, thus connectors can literally perform the same computation as done by components. This gives an advantage to Aesop in modeling patterns that require complex communications e.g. TCP/IP and Remote Procedure Call (RPC) in Client-Server.

UniCon: UniCon provides support for a limited set of built-in types of abstractions (i.e. specialized set of architecture elements) to represent pattern elements. In specific, UniCon supports connector abstractions of type Pipe, ProcedureCall, RPC, RealTimeScheduler, DataAccess, and PLBundler [37]. For instance, when modeling a Pipe-Filter pattern, the connector abstraction for the pipe provides support for specifying the number of connections, input ports, output ports, source roles, and sink roles, etc. Thus, only the existing abstractions available in UniCon can be used to specify constraints and to represent pattern elements. This makes it a weak option for modeling patterns, which are not supported by existing abstractions in UniCon.

xADL: xADL provides five XML (Extensible Markup Language) based tags to represent architecture elements, namely ¡Architecture¿, ¡Component¿, ¡Connector¿, ¡ComponentType¿, and ¡ConnectorType¿ [50]. xADL contains the inherent features of XML, which allow to extend tags for expressing pattern elements. Each tag can be enforced with pattern elements specific constraints. For instance, in a pipe-filter pattern, ComponentType defines nature of filter (e.g. message passing, data computation, data conversion etc.), and ConnectorType defines nature of pipe (e.g. input and output type of parameters). xADL supports type of connections using XML DTDs (Document Type Definitions) [44], which means different kinds of connections to express pattern elements can be used by specifying DTDs. Furthermore, these DTDs can be used to constrain the behavior of interacting pattern elements. For instance, a filter port can define the type of messages it receives using DTDs. Since tags in xADL represent general concepts to express architecture elements, manual work with these tags is required to fully express patterns.

Table 2.1 provides a brief description of the syntax support offered by each ADL for modeling patterns.

ADLs / Patterns	Layers	Pipes and Filters	Blackboard	Client-Server
UML 2.0	Strength: Package metaclass support in UML can be used to group components Weakness: UML Aggregation, Composition and Package structure are not suitable to model all concerns of a layered pattern	Strength: Connector metaclass in UML can be extended to express pipes Weakness: weak support for pipe representation	Weakness: Connectors have fixed interfaces which affects dynamic configuration	Strength: UML profile can be extended to express client-server components and to define constraints on client-server topologies Weakness: UML profile in itself provides weak connector support for complex communication
ACME	Strength: ACME templates can be used to express grouping among components	Strength: Templates can be defined in ACME to express filters, pipes and data flow links	Weakness: Dynamic composition of components and connectors is weakly supported	Strength: Templates can be used to express client-server components and configuration constraints for defining communication links and topologies
Wright	Strength: Roles and glue specification can be used to express layered information flow constraints	Strength: Wright provides roles and glue support for expressing pipes and to define data flow connections among filters	Strength: provides constructs to describe dynamics of the components and provides events support to notify the state change of the components	Strength: Compatibility checking of clients and server is well supported, Deadlock detection is addressed by the use of roles and glues, allows complex topologies, reconfiguration supported, dynamism supported Weakness. Topological constraints not explicitly addressed
AESOP	Strength: Pattern-specific elements and constraints can be expressed by defining and extending sub-types of the generic elements: components, connectors, configuration, ports, roles, bindings, etc. Weakness: Configuration rules not very well supported for dynamic composition			
UniCon	Weakness: Fixed pattern elements specific abstractions is a problem to express layered pattern specific constraints	Strength: Implicit abstractions support for expressing pipes and filters	Strength: Dynamic configuration and analysis supported Weakness: Fixed set of abstractions to represent pattern elements is a problem to define flexible configuration rules	Strength: Rich abstractions to represent communication links supported e.g. connector abstractions for procedure call, RPC, RealTimeScheduler for real-time communication, etc.
xADL	Strength: Grouping structure can be extended to express Layers	Strength: Tags can be extended to express pipes and filters	Strength: Dynamic configuration of architecture elements supported, Events can be used to inform the connected elements about the state change	Strength: A variety of communication protocols can be specified by specifying new kinds of DTDs and tags

Table 2.1: *Syntax Support for Patterns in the ADLs*

2.4.2 Visualization

UML: A number of UML tools have been developed with explicit support for visual software designing e.g. IBM Rational Rose, Rational Software Architect, ArgoUML, etc. However, none of the tools developed for UML specifically focus on modeling architecture patterns. As a solution, few of the UML tools provide visual support to extend UML metamodel elements. For instance, Rational Rose allows user to create stereotypes, which are extensions to UML metaclasses, to model pattern elements. Still, UML tools are weak in providing explicit visualization support to model patterns and it largely depends on the way these UML tools are used to configure architecture elements for modeling patterns.

ACME: ACME has the advantage that with the introduction of ACME studio, which is an

extension to Eclipse, it provides explicit visualization support to model specific patterns. The ACME studio editor provides three views: overview of the files in the project, textual source of the architecture and architecture diagrams with visibility of modeled patterns. To model specific patterns, ACME studio allows one to directly associate pattern elements with their corresponding architecture elements. For example, a component can be created by selecting a pattern type as filter, server, etc. In addition, ACME studio provides visualization support to view pattern elements at both abstract and detail level. For instance, a selected filter can be expanded to view its internal structure of pipes and filters.

Wright: Wright does not provide specific visualization support for modeling patterns.

Aesop: For visual modeling of patterns, Aesop supports a palette of pattern specific architecture elements and an interface that allows tools to manipulate architecture descriptions [35]. The graphical palette represents pattern-specific customized architectural elements for the modeling of architecture patterns. For example, pattern-specific graphical icons can be included in the palette e.g. pipes, filters, server, etc. In addition, Aesop stores architecture descriptions as objects in its object base and external tools can access this object base to provide visual editors for modeling patterns, creation and manipulation of objects, etc. [35]. Furthermore, Aesop provides a coloring scheme to identify mismatched connections. For instance, in a Pipe-Filter pattern, a color can be used to highlight incorrectly attached pipes [56].

UniCon: UniCon provides a specialized set of graphical icons to support traditional patterns like Pipe-Filter, Client-Server, etc. These graphical icons are provided in UniCon's default listing of component and connector types e.g. cloud for abstract binding, pipe, clock for real time communication, etc. For compatibility checking, UniCon provides graphical support to identify mismatched connections. For example, when a connection with mismatched signature is proposed, the editor facilitates including a connector that can translate the calling signature to the declared signature [37].

xADL: xADL benefits from associated XML compliant tools that can be used for visual description of software architecture (e.g. XSV and XML Spy [57]). However, xADL does not provide specific visualization support for modeling architecture patterns and it mainly depends the way these tools are manually used by the architects to express architecture patterns.

Table 2.2 gives a brief description of visualization support offered by each ADL for modeling patterns in general.

ADLs	Pattern modeling support
UML 2.0	Strength: UML tools support visual composition of components and connectors, which can be used for modeling specific concepts of architecture patterns Weakness: UML does not provide explicit support for modeling architecture patterns
ACME	Strength: ACME studio provides explicit visualization support to model few selected patterns Wright Weakness: No specific visualization support provided for modeling architecture patterns
AESOP	Strength: Pattern-specific architecture elements with distinctive colors can be visually created. Pattern elements can be composed for modeling specific patterns.
UniCon	Strength: For a specific list of patterns, UniCon provides good graphical support for modeling patterns and to convert graphical diagrams into textual description Weakness: UniCon provides graphical support for modeling only few patterns.
xADL	Weakness: No specific visualization support provided for modeling architecture patterns

Table 2.2: *Visualization Support for Patterns in ADLs*

2.4.3 Variability

UML: Fixed interfaces, weak connector support and lack of explicit support to express pattern elements is a problem for modeling variability in patterns. Extending UML, as discussed in previous sections, is an explicit way to model pattern variability. However, even the extension to

UML metamodel can address a limited variability in patterns. First, because pattern variability at detail level of design is not addressed at a higher level of abstraction to represent architecture elements as done in UML. Secondly, OCL constraints need to be explicitly addressed for each specific variability issue for modeling patterns. For instance, an OCL constraint restricting no more than two ports attached to a filter will always fail in the operation to add a third port to a specific filter and some sort of extension to OCL description is required.

ACME: ACME defines a weak typing system with a fixed set of types e.g. seven architectural elements of its core ontology and data types of Integer, Boolean, and String [58]. This provides ACME both an advantage and disadvantage in modeling patterns variability. An advantage is that being a standard interchange platform between ADLs, ACME provides a generalized support to represent architecture elements, which is extensible to model variability in patterns. For instance, a filter in Pipe-Filter pattern resembles a generic ACME component with input and output ports. This allows using a filter in all required contexts by considering it as a mere component endowed with the properties of a filter. However, this flexibility in the language has a negative impact on the analysis of modeled variability as no explicit type checking support is provided in ACME.

Wright: Flexible glue specification provides support for modeling pattern-specific variability. The glue specification for connectors allows pattern elements of same type to be represented as logically separate type of entities. For instance, each pipe in a pipe-filter pattern can express its own glue specification to connect with filters. Therefore, a pipe can be connected on one end to a filter and on the other end to a file, while other pipes in the same chain may be connected on both ends to filters. This strong representation of connections among architectural elements gives advantage to Wright in modeling specific variability by providing each architectural connection specific glue specification. Furthermore, rich specification of connector allows distinctively identifying variants of connectors e.g. pipes, procedure calls, etc.

Aesop: As discussed in previous sections, Aesop facilitates creation of environments to define patterns. These pattern definitions are compiled during environment creation time. While modeling patterns in software design, it does not support any kind of variability, which is not included in the original definition of the pattern. For instance, in the pipeline pattern, a filter is always initialized with only one input and one output port. A variability requirement to add a fork in pipeline will always fail in adding a new port. Furthermore, pattern-specific customization of classes requires architect to handle variability constraints at its own with least help from language.

UniCon: UniCon provides a limited set of abstractions to represent pattern elements and connections. This puts a huge constraint in modeling specific type of variability in UniCon as it allows representing connections from only existing types of abstractions. For instance, a procedure call can be replaced with a different connection from only available types of connections.

xADL: xADL defines schemas named: options (optional components, connectors, and links), variants (variant component and connector types), versions (versions in the form of graphs for components, connectors, and interfaces) [50]. These features supported by each schema can be used to model limited variability in patterns. The use of options and variants gives architects freedom to specify pattern elements of different types (e.g. different types of filters) in a single xADL document, and then instantiate any of the pattern elements during architecture design. Furthermore, xADL supports a programming language style type system for specifying pattern elements [50]. Thus, architects can define different variants of the pattern elements as types of component, connector, and interfaces. For instance, a filter type can be extended to specify one or more filters with different properties.

2.4.4 Extensibility

UML: UML is considered weak to represent elements of architecture patterns, which is a drawback to model new patterns as well. However, UML's metamodel can be extended to model new patterns. Medvidovic et al. [46] provides UML extension mechanism with the use of UML metaclasses, which can be effectively used to provide extensibility support to model new patterns. For instance, new stereotypes can be created and constraints specific to new patterns can be applied on these stereotypes.

ACME: ACME allows templates to specify recurring patterns, which is helpful in modeling patterns that come-up even with new syntax definitions. These templates are quite flexible supporting new definition of components and connectors. Furthermore, it allows defining new constraints for interaction among components. However, defining architecture elements in ACME requires following the typing discipline applied in ACME as discussed in previous sections. Its typing discipline with a fixed set of data types has the disadvantage that it does not support connections that require new data types.

Wright: Enriched connector support and flexible properties specification makes Wright a preferable extensible language to model new patterns that heavily rely on communication specification. For instance, the Remoting Error pattern [10] can benefit from glue and protocol specification to detect and handle network failures, server crashes, and un-reliable networking objects, etc.

Aesop: Aesop provides a generic list of elements that can be customized to fulfill the requirements to model new patterns. The principal of sub-typing introduced in Aesop can be used to express new pattern elements as sub-types of generic architecture elements. This makes Aesop an attractive option to model new patterns by defining new pattern specific design environments.

UniCon: UniCon provides support for only built-in component types like module, computation, shared data, filter, process, general etc., and built-in connector types like Pipe, ProcedureCall, DataAccess, etc. [37]. It specifies type 'general' for all other types of components that are not supported by it and provides no extension facility to specify new kind of connectors. This puts a huge constraint on modeling new patterns that demand new compositional elements. The benefit that UniCon offers by providing implicit support for modeling few patterns is questioned by its rigidness to support new type of components and connectors.

xADL: xADL, also called 'extension ADL' [44], shows high promises for extensibility to express newly discovered architecture patterns. xADL use of schemas supports extension to express new types of components, connectors, interfaces, connections and configuration rules. Similar to UML stereotyping extensions described in previous sections, xADL supports extensibility by new tags and attributes. However, extension mechanism of XML itself imposes some restrictions to express pattern elements as it offers a weak support in applying constraints on new pattern elements.

2.5 Related Work

The idea to compare ADLs for their suitability to design software architectures has already been investigated from different viewpoints [16], [51], etc. However, none of the approaches presented so far have specifically focused on comparison of the ADLs for their support to model architecture patterns. Most of the work to date, has focused on the use of mere components and connectors to design software architecture, neglecting the pattern rules for the composition of architecture elements. In our work, we have specifically focused on modeling patterns in few selected ADLs to analyze their support to model patterns.

Medvidovic et al. [16] provide a comparison framework to compare architecture modeling features and tool support offered by a number of ADLs. Their work is focused on components, connectors and their configuration. They highlight the inconsistency with which different ADLs specify semantics to configure components and connectors, and the problems for specifying non-functional properties. Our work is complimentary to this general survey of ADLs, as we focus on the use of patterns to design software architecture.

Shaw et al. [40] analyze patterns for their topology, configuration, data, and control issues. Their work is based on the feature selection among patterns to guide the architects to choose a pattern that is best suited to solve the problem at hand. The framework they propose accommodates patterns in the categories of communicating processes and dataflow networks. They also specify association of specific patterns with their description languages. However, their work is more focused on the selection of patterns to solve the problems, with little attention on challenges to model these patterns in ADLs. Our work is different in the sense that we specifically focus on patterns to relate them with different ADLs to provide a comparison among ADLs for their support in modeling patterns.

In our previous work [15], we have used architecture primitives as an extension to UML metamodel elements to model patterns. Although this work is focused on UML 2.0, the same approach can be used for other ADLs as long as the selected ADL supports the extension mechanism to handle the semantics of the primitives. The key idea in this approach is that the languages that can be extended to facilitate syntactic and semantic of architecture primitives can be used to model pattern variability.

2.6 Conclusions

We have evaluated a few selected ADLs for their support to model architecture patterns. An evaluation framework that looks into syntax, visualization, variability, and extensibility was used to serve this purpose. We find that most of the ADLs specify strong notational, analysis and tool support to design software architectures. Furthermore, some of these ADLs provide inherent support to model patterns but at a detailed level, nearly all of the ADLs fail to capture the rich concepts found in patterns. Furthermore, ADLs differ largely in their scope to model patterns. Few ADLs are popular for modeling patterns due to their specialized nature for providing abstraction support to represent pattern elements. However, none of the ADLs deal with the variability issues for modeling patterns in general. For each ADL discussed in this chapter, some of the strong and weak points were highlighted for their support to model patterns.

We find extension mechanism for some of the ADLs as an effective way for modeling patterns. ADLs, like UML and Aesop, provide a generic list of customizable elements to express pattern specific elements. However, the shortcoming of this approach stems from the use of the ADLs itself. Specifically, the extension mechanism of UML is awkward to use because the extended classes are neither a part of metamodel nor are they model elements [15].

Other than simple pattern representation, ADLs are weak for their accompanying visualization and tool support. Some languages like Aesop, Wright, and UniCon provide tools for type and constraint checking, but their support is limited for the specific use of tools, such as FDR for Wright and RMA for UniCon.

Published as: A. W. Kamal and P. Avgeriou – *"Modeling Architectural Patterns Variants for Designing Software Architectures,"* in proceedings of the European Pattern Languages of Programming (EuroPLoP), 9 - 13 July 2008, Irsee, Germany

Chapter 3

Modeling Architectural Patterns Variants

Abstract

Systematic modeling of architectural patterns is a challenging task mostly because of the inherent pattern variability and because pattern elements do not match the architectural abstractions of modeling languages. In this chapter, we describe an approach for systematic modeling of architectural patterns using a set of architectural primitives and a vocabulary of pattern-specific architectural elements. These architectural primitives can be used as the basic building blocks for modeling a number of architectural patterns. We introduce profiles for the UML2 meta-model to express the architectural primitives. The use of the primitives along with the stereotyping scheme is capable of handling some of the challenges for the systematic modeling of architectural patterns, such as expressing pattern participants in software design.

keywords: Architectural Pattern, Architectural Primitive, Modeling, UML.

3.1 Motivation

Architectural patterns provide solutions to recurring problems at the architecture design level. These patterns not only document 'how' solution solves the problem at hand but also 'why' it is solved, i.e. the rationale behind this specific solution [15]. So far, a huge list of patterns has been documented in the literature [4, 59]. These patterns have been successfully applied to design software in different domains and provide concrete guidelines for modeling the structural and behavioral aspects of software systems. Although at present, the practice of modeling architectural patterns is largely ad hoc and unsystematic, the topic of systematic pattern modeling is receiving increasing attention from researchers and practitioners [49].

In spite of the benefits that patterns offer for solving recurring design problems and the ever-growing list of documented patterns, there is not yet a proven approach for the systematic modeling of architectural patterns and pattern variants in software design. Some architecture description languages (ADLs), such as UniCon [37], Aesop [35], ACME [38], and Wright [42] capture specific concepts for modeling patterns. However, none of the approaches presented so far, for modeling architectural patterns, can effectively express the semantics of architectural patterns [28]. This is because each pattern addresses a whole solution space comprised of different variants of the same pattern, which are difficult to express in a specific ADL. In contrast to ADLs, UML offers a generalized set of elements to describe software architecture but UMLs support for modeling patterns is weak because pattern elements do not match the architectural abstractions provided in UML. In summary, both ADLs and the UML provide only limited support for modeling patterns.

In our previous work [15], we identified a set of architectural primitives. These primitives offer reusable modeling abstractions that can be used to systematically model solutions that

are repetitively found in different patterns. In this chapter, we introduce a few more primitives and use all the primitives discovered during our current and previous work to devise an approach that is capable of systematically modeling architectural patterns in system design. The main contribution of this chapter lies in modeling pattern variants using primitives, identifying pattern aspects that are difficult to express using primitives, and devising a generalized scheme that uses a vocabulary of pattern-specific components and connectors (e.g., pipes, filters) in conjunction with primitives for systematically modeling architectural patterns.

The remainder of this chapter is structured as follows: In Section 3.2 we present our approach for representing patterns and primitives as modeling abstractions, exemplified using an extension of the UML. Section 3.3 briefly introduces the primitives discovered in our previous work while Section 3.4 gives detailed information of the new primitives documented in this chapter. In section 3.5, we give an overview of the relationships between patterns and primitives. Section 3.6 describes the modeling of few selected pattern variants using primitives and a pattern-elements vocabulary. Section 3.7 compares related work and Section 3.8 concludes this study.

3.2 Extending UML to Represent Patterns and Primitives

UML is a widely known modeling language and is highly extensible [49]. There are two approaches for extending UML: extending the core UML metamodel or creating profiles which extend metaclasses. Our work focuses on the second approach where we create profiles specific to the individual architectural primitives. Although this work is exemplified using UML 2.0, the same approach can be used for other modeling languages as long as the selected modeling language supports an extension mechanism to handle the semantics of the primitives. The key idea is that a modeling language can be extended to facilitate semantics of the architectural primitives and that these primitives can then be used to model patterns.

We extend the UML metamodel for each discovered architectural primitive using UML profiles. That is, we define the primitive as extensions of existing metaclasses of the UML using stereotypes, tagged values, and constraints as already discussed in previous chapter.

We chose the UML profiles extension mechanism due to the following reasons:

- A large community of software architects understands UML as a software modeling language. This enables us to use the existing set of UML elements as the basis for extensions. Thus, the time needed to learn a new language and the risks of a novel approach are reduced.
- UML allows the creation of profiles without changing the semantics of the underlying elements of the UML metamodel. Profiles are good enough to serve for this purpose.
- A number of UML tools are available to design software architecture and support profiles out-of-the-box. In contrast, a metamodel extension would require an extension of the tools.

In the architectural primitives, presented in this chapter, we mainly extend the following classes of the UML 2 metamodel to express the primitives:

- *Components* are associated with required and provided interfaces and may own ports. Components use connectors to connect with other components or with its internal ports.
- *Interfaces* provide contracts that classes (and components as their specialization) must comply with. We use the interface meta-class to support provided and required interfaces,

where provided interfaces represent functions offered by a component and required interfaces represents functions expected by a component from its environment.

- *Ports* are the distinct points of interaction between the component that owns the ports and its environment. Ports specify the required and provided interfaces of the component that owns them.

- *Connectors* connect the required interfaces of one component to the provided interfaces of other matching components.

3.3 Architectural Primitives

This section provides an extension to our previous work [15] where we listed nine architectural primitives along with the mechanism to discover primitives in architectural patterns. We have used the same mechanism to discover new primitives in this chapter. We first present five primitives discovered in the Component-Connector view that are repetitively found as abstractions in modeling variants of a number of patterns. Moreover some patterns documented in [7] are used as solution participants of other patterns, hence we consider their modeling solution as primitives and include them in our collection. Subsequently, in the next section, we extend the set of primitives with five new primitives.

Our original set of primitives was comprised of the following [15]:

- *Callback:* A component B invokes an operation on Component A, where Component B keeps a reference to component A in order to call back to component A later in time.

- *Indirection:* A component receiving invocations does not handle the invocations on its own, but instead redirects them to another target component.

- *Grouping:* Grouping represents a Whole-Part structure where one or more components work as a Whole while other components are its parts.

- *Layering:* Layering extends the Grouping primitive, and the participating components follow certain rules, such as the restriction not to bypass lower layer components.

- *Aggregation Cascade:* A composite component consists of a number of subparts, and there is the constraint that composite A can only aggregate components of type B, B only C, etc.

- *Composition Cascade:* A Composition Cascade extends Aggregation Cascade by the further constraint that a component can only be part of one composite at any time.

- *Shield:* Shield components protect other components from direct access by the external client. The protected components can only be accessed through Shield.

- *Typing:* Using associations, custom typing models are defined with the notion of super type connectors and type connectors.

- *Virtual Connector:* Virtual connectors reflect indirect communication links among components for which at least one additional path exists from the source to the target component.

3.4 Description and Modeling Solutions to Architectural Primitives in the Component-Connector View

In this section, we present five primitives that are repetitively found among a number of architectural patterns. For the first selected primitive, we briefly describe the primitive, discuss the issues of modeling the primitive in UML, present UML profile elements as a concrete modeling solution for expressing the primitive, and motivate known uses of the primitive in architectural patterns. For the sake of simplicity, the modeling issues and modeling solutions of remaining primitives are detailed in the Appendix A.

3.4.1 Push-Pull

Context: Push, Pull, and Push-Pull structures are common abstractions in many software patterns. They occur when a target component receives a message on behalf of a source component (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull).

Modeling Issues: Semantics of push-pull structures are missing in UML diagrams. It is difficult to understand whether a certain operation is used to push data, pull data, or both. A major problem in modeling the patterns using Push or Pull in UML is that although Push-Pull structures are often used to transmit data among components, it cannot be explicitly modeled in UML.

Modeling Solution: To properly capture the semantics of Push-Pull in UML, we propose a number of new stereotypes for dealing with the three cases Push, Pull, and Push-Pull. Figure 1 illustrates these stereotypes according to the UML 2.0 profile package, while Figures 2 and 3 depict the notation used for the stereotypes.

The Push-Pull primitive consists of the following stereotypes and constraints:

- *IPush*: A stereotype that extends the Interface metaclass and contains methods that Push data among components.
- *IPull*: A stereotype that extends the Interface metaclass and contains methods that Pull data among components.
- *PushPort*: A stereotype that extends the Port metaclass and is supported by IPush as provided interface and IPull as required interface. This can be formalized using two OCL constraints:

A Push port is typed by IPush as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IPush.baseInterface->exists (j | j=i))
```

A Push port is typed by IPull as a required interface

```
inv:  self.basePort.required->size() = 1
and self.basePort.required->forAll(
   i:Core::Interface |
```

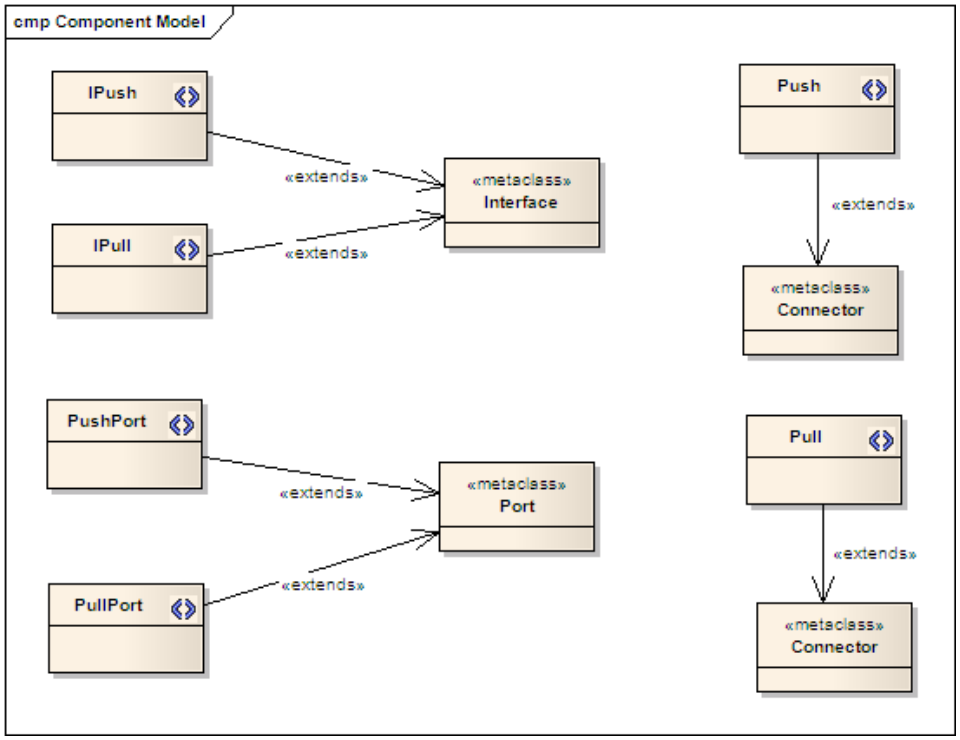

Figure 3.1: *Stereotypes for modeling Push-Pull*

```
IPull.baseInterface->exists (j | j=i))
```

PullPort: A stereotype that extends the port metaclass and is supported by IPush as required interface and IPull as provided interface. This can be formalized using two OCL constraints for the Pull port:

A Pull port is typed by IPull as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IPull.baseInterface->exists (j | j=i))
```

A Pull port is typed by IPush as a required interface

```
inv:  self.basePort.required->size() = 1
and self.basePort.required->forAll(
   i:Core::Interface |
      IPush.baseInterface->exists (j | j=i))
```

Push: A stereotype that extends the Connector metaclass and connects a PushPort with a matching PullPort of another component.

A Push connector has only two ends.

```
inv:  self.baseConnector.end->size() = 2
```

A Push connector connects a PushPort of a component to a matching PullPort of another

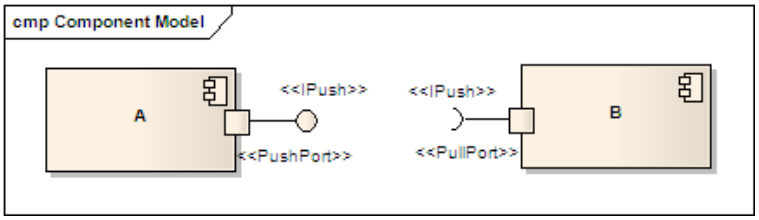

Figure 3.2: *Ports and interfaces to model Push structure from B to A*

component. A PushPort matches a PullPort if the provided interface of the former matches the required interface of the later

```
inv:  self.baseConnector.end->forAll(
e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
   (e1.role->notEmpty() and
      e2.role->notEmpty()) and
         (if PushPort.basePort->exists(p |
            p.oclAsType(Core::ConnectableElement) =
   e1.role)
then
   (PullPort.basePort->exists(p |
      p.oclAsType(Core::ConnectableElement) =
         e2.role) and
            e1.role.oclAsType(Core::Port).required =
               e2.role.oclAsType(Core::Port).provided)
else
   PullPort.basePort->exists(p|
      p.oclAsType(Core::ConnectableElement) =
         e1.role)
endif)))
```

Pull: A stereotype that extends the Connector metaclass and connects a PullPort with a matching PushPort of another component.

A Pull connector has only two ends.

```
inv:  self.baseConnector.end->size() = 2
```

A Pull connector connects a PullPort of a component to a matching PushPort of another component. A PushPort matches a PullPort if the provided interface of the former matches the required interface of the later

```
inv:  self.baseConnector.end->forAll(
e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
   (e1.role->notEmpty() and
      e2.role->notEmpty() ) and
         (if PushPort.basePort->exists(p |
            p.oclAsType(Core::ConnectableElement) =
               e1.role)
```

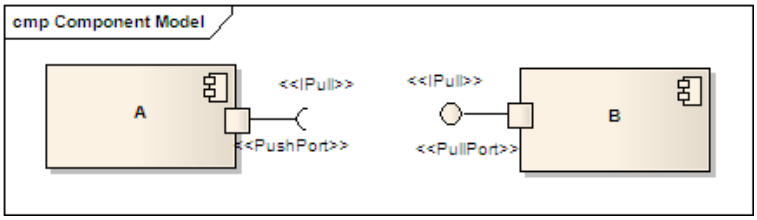

Figure 3.3: *Ports and interfaces to model Pull structure from A to B*

```
        then
(PullPort.basePort->exists(p |
   p.oclAsType(Core::ConnectableElement) =
      e2.role) and
         e1.role.oclAsType(Core::Port).required =
            e2.role.oclAsType(Core::Port).provided)
         else
PullPort.basePort->exists(p|
p.oclAsType(Core::ConnectableElement) =
e1.role)
endif)))
```

Known uses in patterns:

- In the Model-View-Controller [5] pattern, the model pushes data to the view, and the view can pull data from the model.
- In the Pipes and Filters [5] pattern, filters push data, which is transmitted by pipes to other filters. In addition, pipes can request data from source filters (Pull) to transmit it to the target filters.
- In the Publish-Subscribe [5] pattern, data is pushed from a framework to subscribers and subscribers can pull data from the framework.
- In the Client-Server [5] pattern, data is pushed from the server to the client, and the client can send a request to pull data from the server.

3.4.2 Virtual Callback

Context: Consider two components are connected via a callback mechanism. In many cases the callback between components does not exist directly, rather there exist mediator components between the source and the target components. Such information should be represented at the design level. For instance, in the MVC pattern, a model may call a view to update its data but this data may be rendered first by the mediator components before it is displayed on the GUI.

Modeling Issues: The virtual relationship is an important aspect to show collaborating elements. The standard UML supports connector or association links to model virtual relationships. However, such a relationship cannot be made explicit in standard UML as it may become difficult to determine which components have subscribed to other components to be called back virtually.

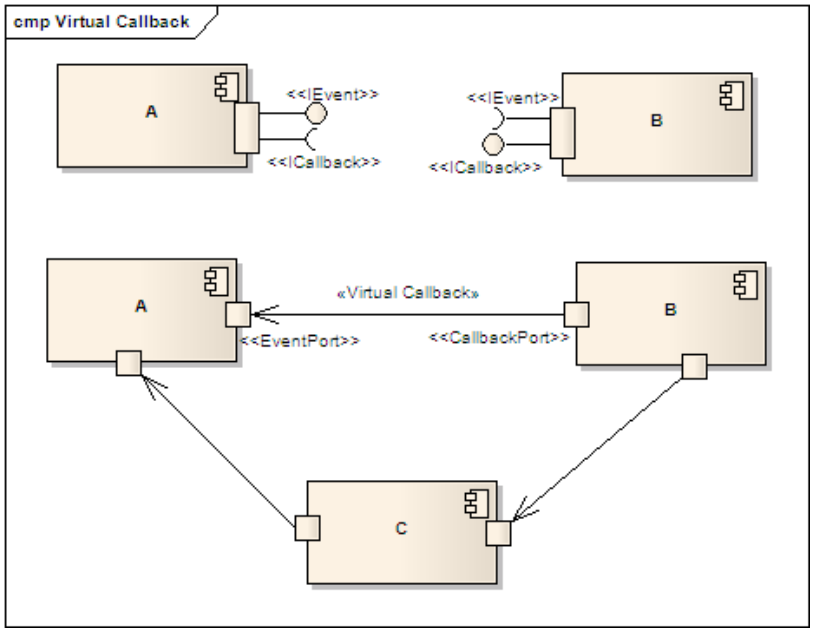

Figure 3.4: *The notation of the stereotypes in Virtual Callback Modeling*

Modeling Solution: To capture the semantics of Virtual Callback properly in UML, we extend the Callback [15] primitive with constraints that a virtual callback can only be used between two components where there is a path of components and connectors that links A to B using following constraints:

To capture the semantics of callback primitive properly in UML, we use the following stereotypes: VirtualCallback, EventEnd, and CallbackEnd. The VirtualCallback extends the Connector metaclass while the EventEnd and CallbackEnd extend the ConnectorEnd metaclass where the EventOccurence takes place at the sender component (EventEnd) while the EventExecution takes place at the receiver end (CallbackEnd).

Known Uses in Patterns:

- In the MVC [5] pattern, the view and model components may communicate to each other virtually using callback operation.
- In the Observer [59] pattern, the subjects may observe the target objects virtually.
- In the Publish-Subscribe [5] pattern, the publishers may callback subscribers virtually.

3.4.3 Delegation Adaptor

Context: This primitive converts the provided interface of a component into the interface the clients expect. The Delegation Adaptor primitive is a close match to the Object Adaptor [59] pattern.

Modeling Issues: Adaptors shield the underlying system implementation from its surroundings. However, adaptors can not be explicitly modeled using the architectural abstractions present in UML as their task is more focused on conversion rather than computation.

Modeling Solution: To capture the semantics of Adaptor properly in UML, we propose the following new stereotypes: AdaptorPort extends the Port metaclass and is typed by the IAdap-

tor as provided interface and IAdaptee as required interface. Both the IAdaptor and IAdaptee stereotypes extend the Interface metaclass.

Known Uses in Patterns:

- In the Layers [5] pattern, the adaptor supports the separation of explicit interface of a layer from its implementation.
- In the Broker [5] pattern, the adaptor translates the messages coming from remote services to the underlying system.
- In the Microkernel [5] pattern, the adaptor is used to map communication between external and internal servers.
- In the Proxy [59] pattern, the adaptor is used to separate the interface from the implementation.

3.4.4 Passive Element

Context: Consider an element is invoked by other elements to perform certain operations. Passive elements do not call operations of other elements.

Modeling Issues: UML components do not structurally differentiate between active and passive elements. Such a differentiation is important to understand clearly the responsibility of individual elements in the design.

Modeling Solution: To capture the semantics of Passive Element properly in UML, we use the following new stereotypes: PElement extends the Component metaclass and attaches the PassivePort. The IPassive stereotype extends the Interface metaclass and types the PassivePort, which extends the Passive metaclass.

Known Uses in Patterns:

- In the Pipes and Filters [5] pattern, the passive filter cannot pull or push data to its neighboring filters.
- In the MVC [5] pattern, the passive view only receives or displays data to the user and does not invoke any operation on a model or controller elements.
- In the Client-Server [5] pattern, the passive server does not invoke any operation on client-side and responds only to the client requests.

3.4.5 Interceder

Context: Sometimes certain objects in a set of objects cooperate with several other objects. Allowing direct link between such objects can overly complicate the communication and result in strong coupling between objects [59]. To solve this problem, Interceder components are used.

Modeling Issues: Interceder components are typically involved in decoupling components and store the collective behavior of interacting components. The structural representation of mediator components in UML diagrams is hard to understand.

Modeling Solution: To capture the semantics of Interceder primitive properly in UML, we propose following new stereotypes: Incdr, IncdrPort, and IFIncdr. Incdr extends the Component metaclass and attaches IncdrPort. IncdrPort extends the Port metaclass and is typed by the provided interface IFIncdr.

Known Uses in Patterns:

- In the PAC [5] pattern, a controller is used to intercede communication between agents in the PAC hierarchy.
- In the Microkernel [5] pattern, an interceder component receives requests from external server and dispatches these requests to one or more internal servers.
- In the Reflection [5] pattern, the meta level components intercede communication by providing interfaces to facilitate modification in underlying components.

3.5 The Pattern-Primitive Relationship

Architectural patterns and architectural primitives are complementary concepts. Modeling patterns in a system design is applying one of the alternate solutions to solve specific problems at hand [5] where as primitives serve as the building blocks for expressing architectural patterns. In this context, patterns offer general solutions while primitives offer relatively more specific solutions. Similar to the selection of architectural patterns among complementary patterns, primitives might also need to be selected among complementary primitives, e.g., based on the system requirements you might choose either Shield or Indirection. Such a decision to select the appropriate primitive involves the context in which the pattern is applied, and the specific solution variant addressed by the pattern. Moreover, certain primitives can be used in combination with other primitives. For example, the Callback and Push-Pull primitives can work in conjunction to serve a common purpose.

Table 1 provides a patterns-to-primitives mapping, which is based on the primitives discovered so far in our work. The detailed discussion about the discovery of each primitive in the related patterns is already documented in the Known Uses in Patterns subsections of our current and previous work (see Section 3 and [15] for details). The intention is to use the pool of all available primitives to model several architectural patterns. However, the mapping from patterns to primitives is not one-to-one: rather different variants of the patterns can be modeled using a different combination of primitives. Thus, the decision to apply a specific primitive for modeling patterns lies with the architect who selects primitive(s) that best meet the needs to model the selected pattern(s).

The issues addressed above directly deal with the traditional challenge of modeling pattern variability. The solution variants entailed by a pattern can be applied in infinite different ways and so is the selection of primitives for modeling pattern variants. More important is that whichever pattern variant is applied in system design, it should address the solution clearly with structural and semantic presence. Using our primitives allows an architect to apply a near infinite solution variants with certain level of reusability. Such a reusability support also depends on the context in which the pattern is applied as in some cases extra constraints or missing pattern semantics may be required.

3.5.1 Expressing Missing Pattern Semantics in UML

An important aspect of modeling architectural patterns is the explicit demonstration of patterns in system design and support for automated model validation. Such a representation helps in better understanding of the system by allowing the user to visualize and validate the patterns. The primitives described above capture recurring building blocks found in different patterns. However, it may be the case that certain pattern aspects of a specific solution variant may not be fully expressed by the existing set of primitives. Therefore, for expressing missing pattern semantics that are not covered by the primitives, we provide support to the user

Patterns / Primitives	Callback	Indirection	Grouping	Layering	Aggregation Cascade	Composition Cascade	Shield	Typing	Virtual Connector	Push	Pull	Virtual Callback	Adaptor	Passive Element	Control	Interceder
Active Repository [4]	*										*	*				
Adaptor [59]		*						*					*			
Broker [5]		*	*				*			*						
Cascade [59]					*	*										
Client Server [5]	*		*	*						*	*	*	*			
Component and Wrapper [4]		*									*					
Composite [59]					*	*										
Facade [59]		*	*				*									
Event [59]	*															
Explicit Invocation [59]	*															
Indirection Layers [4]			*	*	*			*		*	*					
Interceptor [59]	*															
Interpreter [59]		*									*					
Knowledge Level [4]								*								
Layered System [4]		*			*					*	*		*			
Layers [5]			*	*			*			*	*				*	
Message Redirector [4]		*					*									
Microkernel [5]		*										*				
MVC [5]	*									*	*	*			*	
Observer [59]	*										*					
Object System Layer [4]					*		*	*								
Organization Hierarchy [4]					*	*										
PAC [5]	*														*	*
Pipes and Filters [5]										*	*			*		
Proxy [5]		*					*									
Publish Subscribe [5]	*									*	*	*	*			
Reactor [59]	*															
Reflection [5]				*								*				
Remote Proxy [4]									*							
Remoting Patterns [10]							*		*							
Type Object [4]								*								
Virtual Machine [4]		*	*				*									
Visitor [59]	*															
Wrapper Facade [4]		*									*					

Table 3.1: *Patterns to Primitives mapping (∗: found in documented pattern)*

with a vocabulary of design elements that can be used alongside with the primitives to fully express pattern semantics such as pipes, filters, client, server etc. For this purpose, we define few stereotype in UML with known semantics of the selected architectural patterns. For instance, a component can be stereotyped as filter and a connector can be stereotyped as pipe. The stereotyping scheme presented here is further complimented by using these stereotypes for modeling the example patterns in the next section.

The use of pattern-specific design elements for expressing pattern variants has a number of significant benefits. First, it offers reusability support for expressing patterns in system design. The well-known properties entailed by documented pattern variants can be reapplied in system design as a solution to new problems. Second, this makes it easier for a stakeholder to understand design of the system. For example, the use of design vocabulary to express pipes

and filters in system design makes an architecture more explicit to understand and the way different architectural elements fit in the structure. Third, it offers a good support for automated model validation by ensuring that selected patterns are correctly applied in a system design. All of these three benefits compliment our use of primitives for modeling patterns. The intention is that though primitives offer good reusability and model validation support, as advocated in our current and previous work [15], the stereotyping scheme presented in this section makes the story complete for the systematic modeling of architectural patterns and pattern variants.

3.6 Modeling Architectural Patterns Using Primitives

In this section, we use the primitives and stereotyping scheme described in the previous sections to model specific pattern variants. The patterns modeled in this section are specialization to the patterns documented in POSA [5] and hence are called pattern variants. We do not claim to cover all the variability aspects of the selected patterns. However, an effort to describe selected pattern variants using primitives provides a solid base for modeling unknown pattern variants as well. To serve this purpose, we have selected three traditional architectural patterns namely the Layers, Pipes and Filters, and Model-View-Controller (MVC). We use the following guidelines to model each selected pattern variant:

- A brief description of selected pattern variants
- Mapping selected pattern variants to the list of available primitives
- Highlight the issues in modeling pattern variants using primitives
- Use stereotyping scheme to capture the missing pattern semantics.

3.6.1 Pipes and Filters

The Pipes and Filters pattern consists of a chain of data processing filters, which are connected through pipes. The filters pass the data output to the adjacent filters through pipes. The elements in the Pipes and Filters pattern can vary in the functions they perform e.g. pipes with data buffering support, feedback loops, forks, active and passive filters etc. The primitives discovered so far address many such variations for systematically modeling Pipes and Filters pattern. However, certain aspects of the Pipes and Filters pattern may not be fully expressed by the primitives e.g. feedback loops, forks, etc. The requirements we consider in this section for modeling the specific Pipes and Filters pattern variant are: a) filters can push or pull data from the adjacent filters; b) filters can behave as active or passive elements; and c) feedback loop.

At first, we map the selected Pipes and Filters pattern variant to the list of available primitives. We select the Push, Pull, and Passive Element primitives from the existing pool of primitives. The rationale behind the selection of these primitives is as follows:

- The Push and Pull primitives are used to express the pipes that transmit streams of data between filters.
- The filters that are not involved in invoking any operations on their surrounding elements are expressed using the Passive Element primitive.

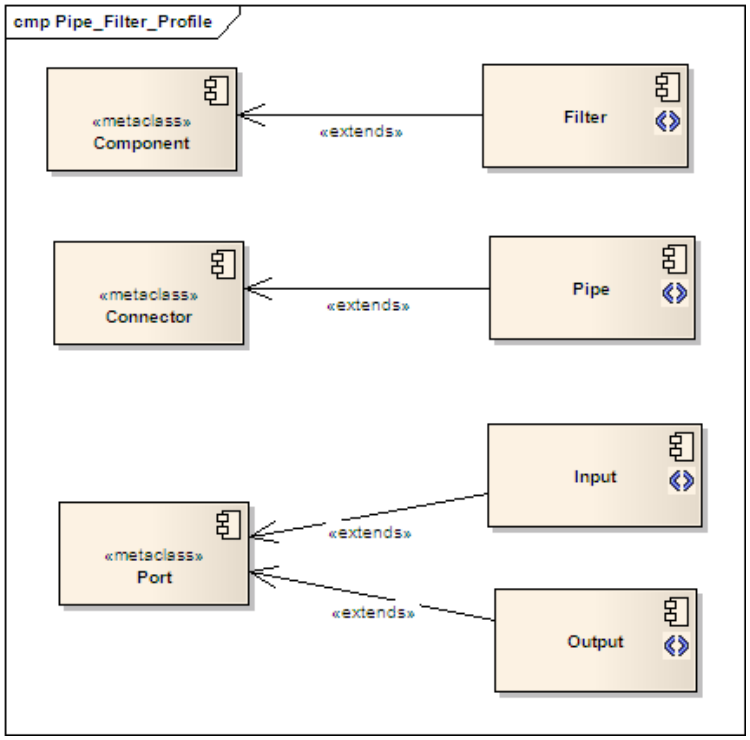

Figure 3.5: *UML stereotypes for expressing Pipes and Filters pattern participants*

Missing Semantics: As described in section four, the challenge to model missing pattern semantics is solved by stereotyping UML elements. In the current example, the selected primitives are sufficient to express the Push, Pull, and Passive Elements in the Pipes and Filters pattern. However, we identify that the feedback loop cannot be fully expressed using the existing set of primitives. The existing primitives can express that the data is pushed or pulled between the filters but this does not express the presence of a feedback structure. Similarly, the semantics of the Pipe and Filter elements are not applied using the existing set of primitives.

Additional Stereotypes: We apply the Feedback stereotype on the Push primitive to capture the structural presence of feedback loop in the Pipes and Filters pattern. Such a structure represents that the data is pushed from one filter to another filter using the feedback loop. The original Push primitive, as described in section four, extends the UML metaclasses of connector, interface, and port. While the feedback stereotype further specializes the Push primitive by labeling it as Feedback. The introduction of feedback stereotype does not introduce new constraints nor affects the underlying semantics of the Push primitive. Figure 5 shows the stereotypes used for expressing Pipes and Filters pattern.

Feedback: A stereotype that is applied to the Push primitive for expressing the Feedback structure in the Pipes and Filters pattern variant. Feedback stereotype extends the Connector metaclass of UML.

The second stereotype named Filter that we use from the existing vocabulary of design elements is defined as follows:

Filter: A stereotype that extends the Component metaclass of UML and attaches input and output ports.

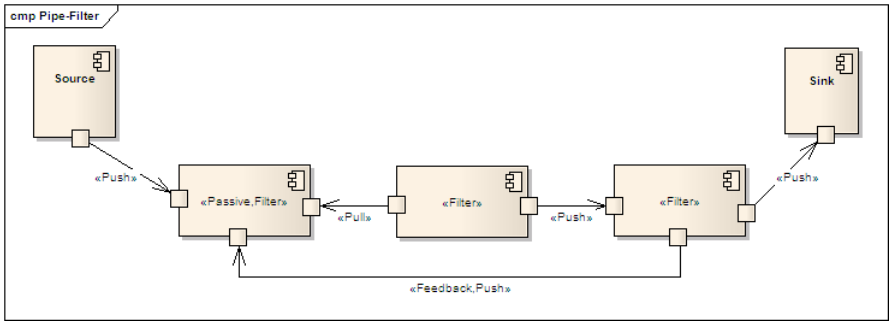

Figure 3.6: *Modeling Pipes and Filters Pattern Variant Using Primitives*

A Filter component is formalized using the following OCL constraints:
An Input port is typed by Iinput as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      Iinput.baseInterface->exists(j | j = i))
```

An Output port is typed by Ioutput as a required interface

```
inv:  self.basePort.required->size() = 1
   and self.basePort.provided->forAll(
      i:Core::Interface |
         Ioutput.baseInterface->exists(j | j = i))
```

The third stereotype that we use from the existing vocabulary of design elements is Pipe that is defined as follows:
Pipe: A stereotype that extends the Connector metaclass of UML and connects the output port of one component to the input port of another component.
A Pipe is formalized using following OCL constraints:

```
inv:  self.baseConnector.end->size() = 2
```

As shown in figure 6, the first filter in the chain works as a passive filter and does not invoke any operations on its surrounding filters. While the second filter is an active filter, which pulls data from the passive filter and after processing pushes this data to the next filter in the chain. The third filter in the chain sends data back to the passive filter for further processing, and sends the final processed data to the sink.

3.6.2 Model-View-Controller

The structure of the MVC pattern consists of three components namely the Model, View, and Controller. The Model provides functional core of an application and notifies views about the data change. Views retrieve information from the Model and display it to the user. Controllers translate events into requests to perform operations on View and Model elements. Usually a change propagation mechanism is used to ensure the consistency between the three components of the MVC pattern [5].

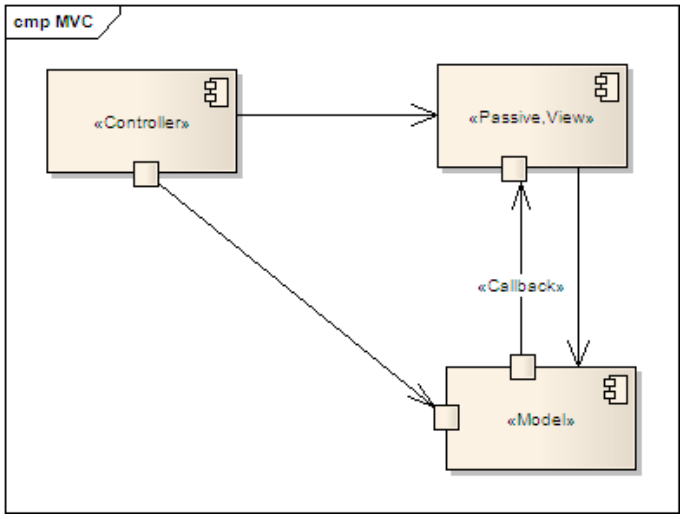

Figure 3.7: *Modeling MVC Pattern Using Primitives*

As a first step, we map the MVC pattern to the list of available primitives as shown in the table in section four. We select the Callback, Passive Element and Control primitives for modeling the MVC pattern. The rationale behind the selection of these primitives is as follows:

- The View subscribes to the model to be called back when some data change occurs and works as passive object by not invoking any operation on the Model.

Missing Semantics: However, not every aspect of the MVC pattern can be modeled using the existing set of primitives. For instance, the Model, View, and Controller components are not mapped to any primitives discovered so far. Keeping in view the general nature of these components, there is a need to provide reusability support by including these three pattern elements in the existing vocabulary of design elements.

Additional Stereotypes: As described above, despite the reusability support offered by the selected primitives, the MVC pattern semantics are not structurally distinguishable. We use the following three stereotypes from the existing set of design elements:

Model: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Controller and View components.

Controller: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Model and View components.

View: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Model and Controller components.

As shown in Figure 7, the Controller receives input and translates it into requests to the associated model using the Control primitive. While, the Model calls back View when a specific data change occurs.

3.6.3 Layers

The Layers pattern groups elements at a certain level of abstraction where lower layers provide services to the adjacent upper layer. Such a structure is used to reduce dependencies between objects in different Layers. As a first step, we map the Layers pattern to the list of available primitives and select the Layering primitive. The rationale behind the selection of this primitive is as follows:

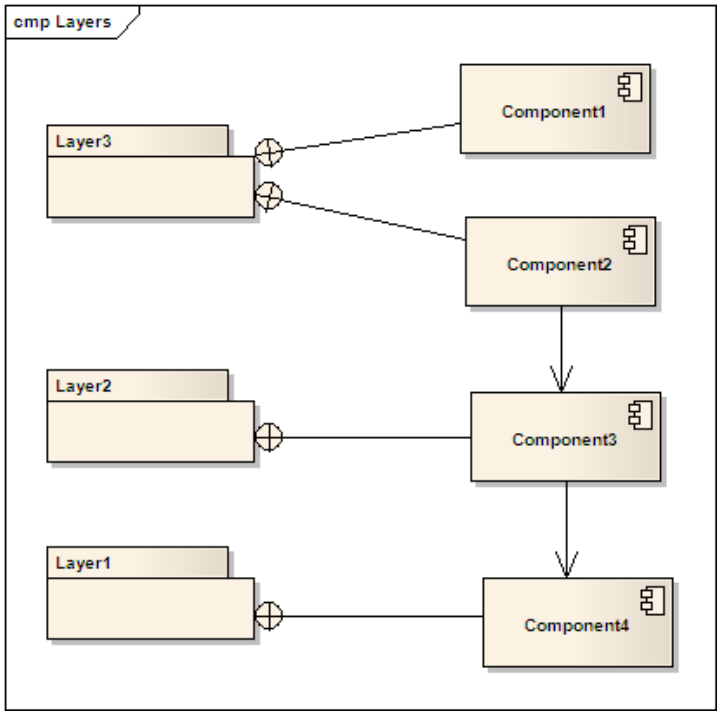

Figure 3.8: *Modeling Layers Pattern Using Primitives*

- Components are members of specific layers where each lower layer provides services to the adjacent upper layer
- A component can only be a member of one layer

Missing Semantics: In the Layers pattern, the high-level functions implementation relies on the lower level ones. Such system requires horizontal partitioning where each partition carries operations at a certain level of abstraction. As each layer in the Layers pattern is a virtual entity so it cannot exist without the presence of at least one component. Moreover, the upper layers cannot bypass the layers for using services in the bottom layers i.e. in Figure 8, the group members from layer3 can call components in layer2, but not into layer1.

Additional Stereotypes: Almost all structural characteristics of the Layers pattern are modeled using the Layering primitive as shown in Figure 8. Using the layering primitive, the constraints assure that within an individual layer all component work at the same level of abstraction and no component belongs to more than one layer at any time. Moreover, no additional stereotyping of UML elements is required to model this specific variant of the layers pattern.

3.7 Tool Support

We advocate that the practical use of such a novel approach requires the presence of a modeling tool that allows modeling of primitives and specialized pattern participants for designing software architecture. The Primus tool [60] has been developed to provide a practical implementation of our ongoing research work. The tool lets a user to effectively define, model, and validate primitives and specific pattern variants within a software architecture. It facilitates the definition of primitive-specific UML profiles. This lets an architect to apply the defined profiles in several different design situations for modeling different patterns.

The OCL used in this chapter for constraining UML elements is a notational language for analysis and design of software systems. It is a part of the UML standard and thus usually used with UML models. With OCL it is possible to define exact constraints and queries without the ambiguities of a programming languages and without the complexities of mathematical definitions. Applying constraints on elements within a model consists of defining a constraint that a certain kind of element must adhere to like number of ports attached to a component. In the Primus tool, OCL is used to check whether such constraints are met or violated. The Primus tool verifies the presence of primitives and pattern variants in the Component and Connector view by collecting the primitives and pattern variants and checking all related constraints. The detailed discussion about the working of the tool is presented in Appendix B.

3.8 Related Work

The approach described in this chapter is based on our previous work [15] where we present an initial set of primitives for modeling architectural patterns. However, the idea to use primitives for software design is not novel and has been applied in different software engineering disciplines [61]. The novelty of our work lies in the use of primitives for systematically modeling architectural patterns, which has not be addressed before.

Using different approaches, a few other researchers have been working actively on the systematic modeling of architectural patterns [35]. Garlan et al. [35] proposes an object model for representing architectural designs. The authors characterize architectural patterns as specialization of the object models. However, each such specialization is built as an independent environment, where each specialization is developed from scratch using basic architectural elements. Our approach significantly differs in a way that our focus is on reusing primitives and pattern elements and only where required we extend the primitives and pattern elements to capture the missing pattern semantics.

Simon et al. [62] extends the UML metamodel by creating pattern-specific profiles. The work by Simon et al. maps the MidArch ADL to the UML metamodel for describing patterns in software design. However, this approach does not address the issue of modeling a variety of patterns documented in the literature rather manual work is required to create profiles for each newly discovered pattern. Our approach distinctively differs from this work as we focus on describing a generalized list of patterns using the primitives.

Mehta et al. [61] propose eight forms and nine functions as basic building blocks to compose pattern elements. Their approach focuses on a small set of primitives for composing elements of architectural styles. Our approach is different in the sense that we offer a more specialized set of primitives that are captured at a rather detail level of abstraction. Moreover, we use vocabulary of pattern elements in parallel to architectural primitives to capture the missing semantics of architectural patterns.

3.9 Conclusions

Using architectural primitives and pattern-specific design elements vocabulary in combination offers a systematic way to model patterns in system design. We have extended the existing pool of primitives with the discovery of five more primitives. With the help of few examples, we show

an approach for modeling architectural pattern variants using primitives. The scheme to use stereotyping in conjunction with primitives offers: a) reusability support by providing vocabulary of design elements that entail the properties of known pattern participants; b) automated model validation support by ensuring that the patterns are correctly modeled using primitives; and c) explicit representation of architectural patterns in system design.

To express the discovered primitives and design elements vocabulary, we have used UML2.0 for creating pattern-specific profiles. As compared with the earlier versions, UML2.0 has come up with many improvements for expressing architectural elements. However, we still find UML as a weak option in modeling many aspects of architectural patterns e.g. weak connector support. As a solution to this problem, the extension mechanisms of the UML offers an effective way for describing new properties of modeling elements. Moreover, the application of the profiles to the primitives allows us to maintain the integrity of the UML metamodel. By defining primitive-specific profiles, we privilege a user to apply selective profiles in the model.

Published as: A. W. Kamal and P. Avgeriou – *"Modeling Architectural Patterns Behavior Using Architectural Primitives,"* proceedings of the 2nd European Conference on Software Architecture (ECSA), pp. 164 - 179, September 29 - October 1, 2008, Pafos, Cyprus, Springer LNCS 5292.

Chapter 4

Modeling Architectural Patterns Behavior Using Architectural Primitives

Abstract

Architectural patterns have an impact on both the structure and the behavior of a system at the architecture design level. However, it is challenging to model patterns behavior in a systematic way because modeling languages do not provide the appropriate abstractions and because each pattern addresses a whole solution space comprised of potentially infinite solution variants. In this chapter, we advocate the use of architectural primitives for systematically modeling architectural patterns in the behavioral view. These architectural primitives are found among a number of architectural patterns and serve as the basic building blocks for modeling patterns behavior. The main contribution of this work lies in the discovery of architectural primitives, defining architectural primitives using UML, and capturing the missing pattern semantics by using UMLs stereotypes.

4.1 Introduction

Architectural patterns provide solutions to recurring design problems that arise in a specific context [41] [59]. These patterns propose a particular structure and behavior that can be tailored to the specific needs of the problem at hand [63] [5]. The solution of an architectural pattern is a model; applying the pattern results in incorporating that model into the software architecture of a specific system. One of the most significant aspects of modeling architectural patterns is the patterns' behavior, which are mostly represented as scenarios that define the run-time actions of the patterns [5]. Such a run-time behavior is vital for the pattern implementation as it shows the way 'pattern participants' collaborate and communicate with each other to express a pattern. We use the term 'participants' to mention the modeling elements that work in association to express architectural patterns. Unfortunately, modeling architectural patterns' behavior in a systematic way remains a challenging task mostly due to the following reasons:

- Pattern participants do not match the architectural abstractions of commonly used modeling languages.
- Architectural patterns' behavior can potentially be modeled in infinite different ways to balance the forces related to the problem at hand.

Architecture Description Languages (ADLs) (e.g. ACME [38] or Wright [42]) and UML [64] have traditionally been used for modeling architectural patterns. Few of these languages focus

specifically on modeling patterns' behavior while few others provide general architectural abstractions that can be extended to express patterns. UML is one such widely known modeling language that offers a generalized set of interaction elements to describe behavioral aspects of software architecture. However, both ADLs and UML provide only limited support for modeling patterns [28] because the architectural abstractions provided by these languages do not match the pattern participants and because they do not provide mechanisms for modeling the infinite variability of pattern behavior.

In our previous work, we have identified a set of architectural primitives in the Component-Connector view [15] and the Process Flow view [65]. We consider the primitives as key participants in modeling patterns and use them as the fundamental modeling elements to express a pattern in system design. These primitives offer reusable modeling abstractions that can be used for systematically modeling pattern variants. In this chapter, we extend our work by focusing on architectural primitives in the behavioral view. We show how few primitives, which are already used for modeling patterns in the structural view, can be used for modeling patterns in the behavioral view as well. We illustrate our approach by presenting how the behavior of three typical architectural patterns can be modeled with the help of these new primitives. Furthermore, since primitives alone do not capture the entire semantics of the patterns, we show how to identify the missing semantics and express them through a vocabulary of pattern-specific objects and messages.

The remainder of this chapter is structured as follows: in Section 4.2, we motivate our choice of selecting UML's collaboration diagram for modeling patterns' behavior. In Section 4.3, we present our approach for representing patterns and primitives as modeling abstractions using an extension of the UML. Section 4.4 gives detailed information of the primitives discovered during our work. In Section 4.5, we use primitives and a vocabulary of design elements, for modeling three selected patterns. Section 4.6 elaborates on related work and Section 4.7 concludes this study.

4.2 The Unified Modeling Language in the Behavioral View

Although any modeling language can be used for modeling architectural primitives as long as the selected modeling language supports an extension mechanism to handle the semantics of the primitives, the UML is our choice in this work. The motivation behind the selection of UML is: a) UML is a widely known de facto modeling language; b) UML provides explicit extension mechanisms; and c) UML supports a variety of diagrams for describing the behavioral aspects of software architecture, such as Use case, Sequence, Collaboration, Statechart, and Activity. Each of these diagrams serves specific purposes to describe software design, which at times overlap with each other. These diagrams use particular UML modeling elements, which can be extended to meet the specific needs of modeling a system. In this chapter, the requirements that we consider for modeling patterns' behavior are as follows:

- *Pattern elements operations:* The operations performed by pattern participants show the true essence of pattern behavior. The operation parameters, return values, and operation type should be represented in the design.

- *Relationships among pattern elements:* The relationships define the nature of interactions performed by the objects, such as the order of occurrence of the operations, multiplicity, and direction of flow etc.

- *Pattern behavior in response to user/system interaction:* Capturing the behavior of pattern

participants that can explain the major dynamics of the pattern when a specific event or user/system action takes place.

Depending on the purpose, the UML supports a variety of diagrams for modeling different aspects of system behavior. A brief description of each UML diagram for modeling system behavior and their comparison to the requirements listed above is given as follows:

- *Use Case Diagrams* describe the interaction between actors - who initiate the action - and the system. The interaction is usually described using a sequence of steps. Use cases are usually defined at a higher level where the system design is considered as a black box, and emerges from the requirements used for designing the system. The use case diagrams, being at a higher level of abstraction, are not a close match to the requirements listed above because our focus lies on detail level interactions and operations among pattern participants.

- *Sequence diagrams* use objects, events, and arrows to depict scenarios by exchanging messages between objects when a specific event occurs. They usually show the execution of a typical example. Sequence diagrams are a close match to the requirements listed above as they show the sequence of operations entailed by the architectural patterns, occurrence of events to invoke specific operations, and use messages to show the interaction among pattern participants.

- *Statechart diagrams* show interactions with other objects inside or outside the system. A state shows the execution of a specific function when an event occurs. State diagrams are more focused on transition of states among objects while our focus lies on interaction among objects, which makes these diagrams a weak option for modeling patterns' behavior in context of the requirements listed above.

- *Collaboration diagrams* depict scenarios as flow of messages. Collaboration diagrams are very similar to sequence diagrams. However, an obvious difference is that collaboration diagrams show the teamwork of messages while sequence diagrams shows the stepwise execution of messages. Similar to the sequence diagrams, we consider collaboration diagrams as a close match to our work since collaboration diagrams can show the operations taking place between the pattern participants, the relationship, and occurrence of specific events.

- *Activity Diagrams* show an operation that is invoked when a specific event occurs. The activity diagram focus on using threads for the transfer of control and data among objects and hence more often used for synchronization checks [64]. These diagrams too are not a close match to the requirements listed above, as activity diagrams do not explicitly show the relationships and interactions among pattern participants.

Thus, we focus on capturing the interaction mechanism between pattern participants using either the sequence diagrams or collaboration diagrams. While sequence diagrams are more restricted to time-bound occurrence of events, the collaboration diagrams are the best choice in this work, which rely on interactions and relationships among objects in a time-independent manner. However, both types of these diagrams are comparative in nature and can be converted from one form to the other.

4.3 Extending UML to Represent Patterns and Primitives

UML is a widely known modeling language and is highly extensible [64]. There are two approaches for extending UML: extending the core UML metamodel or creating profiles by extending metaclasses. Our work focuses on the second approach, i.e. we create profiles specific to the individual architectural primitives. To capture the missing patterns semantics and to express the discovered architectural primitives, we extend the UML metaclasses using UML profile mechanism. That is, we define the primitives and pattern participants as extensions of existing metaclasses of UML using stereotypes and constraints as follows:

- *Stereotypes:* We use stereotypes to extend the properties of existing UML metaclasses. For instance, the Message metaclass is extended to generate a variety of primitives and specialized messages between pattern participants.
- *Constraints:* We use the Object Constraint Language (OCL) [66] to place additional semantic restrictions on extended UML elements. For instance, constraints can be defined on associations between objects, navigability, direction of communication, etc.

4.3.1 The UML 2 metamodel

For the primitives presented in this chapter, we mainly extend or use the following metaclasses of the UML 2.0 interaction metamodel to express the primitives:

- *Messages* are used to perform operations on the objects. Messages define a specific kind of communication in an interaction and connect the MessageEnds, which store references to the adjacent objects that need to be connected.
- *Interaction* provides connection between connectable elements using message ends. It uses namespace to store the sequence of operations taking place in the collaboration diagrams.
- *MessageEnd* connects the source object to the target object, where the source and target objects own the message ends.

We have also used the following UML metaclasses in order to express the constraints on UML metamodel:

- *EventOccurence* is a specialization of the MessageEnd. The message operations use the MessageEnds to send and receive events.
- *ExecutionOccurence* is represented by two event occurrences, the start event occurrence and the finish event occurrence.

4.4 Architectural Primitives

This section presents a continuation to our previous work where we have listed several architectural primitives in Component-Connector view [15] and the Process Flow view [65]. In this section, we present seven primitives discovered in the behavioral view that are repetitively found as abstractions in a number of patterns. The aim of our work is to capture common recurring solutions at an abstraction level that can be used to model architectural patterns' behavior, hence providing a better reusability and systematic support to model patterns. Following, we list the primitives discovered during our work and present the UML profile elements as a concrete modeling solution for expressing these primitives.

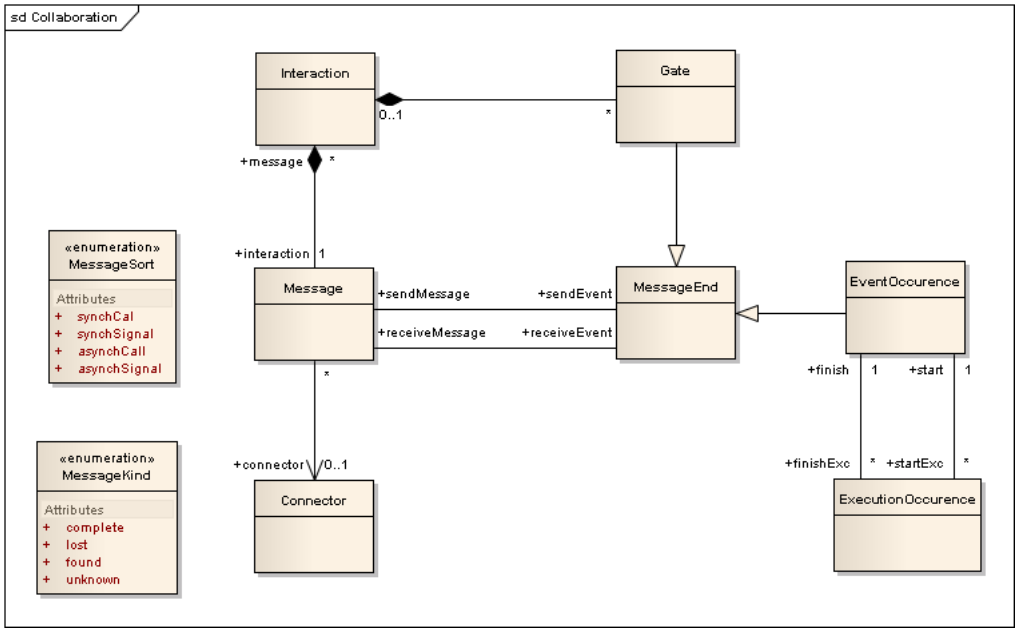

Figure 4.1: *Part of the UML Interaction metamodel used for defining primitives*

4.4.1 Documenting an Architectural Primitive: Push-Pull

Textual Description: Push, Pull, and Push-Pull structures are common abstractions in many software patterns. They occur when a target object receives a message sent by a source object (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull).

Known uses in patterns:

- In the Model-View-Controller [5] pattern, the model pushes data to the view, and the view can pull data from the model.
- In the PIPE-FILTER [5] pattern, filters push data, which is transmitted by the pipes to other filters and even pipes can request data from source filters (Pull) to transmit it to the target filters.
- In the PUBLISH-SUBSCRIBE [5] pattern, data is pushed from the framework to subscribers and subscribers can pull data from the framework.
- In the CLIENT-SERVER [5] pattern, data is pushed from the server to the client, and the client can send a request to pull data from the server.

Modeling Issues: Semantics of the push-pull structure is missing in UML diagrams. It is difficult to understand whether a certain operation is used to push data, pull data, or both. A major problem in modeling the above listed patters in UML is that although a Push-Pull structure is often used to transmit data among objects, it cannot be explicitly modeled using UML interaction diagrams.

Modeling Solution: To capture the semantics of Push-Pull properly in UML, we propose a number of new stereotypes for dealing with the three cases: Push, Pull, and Push-Pull. Figure 2 illustrates these stereotypes according to the UML 2.0 interaction model.

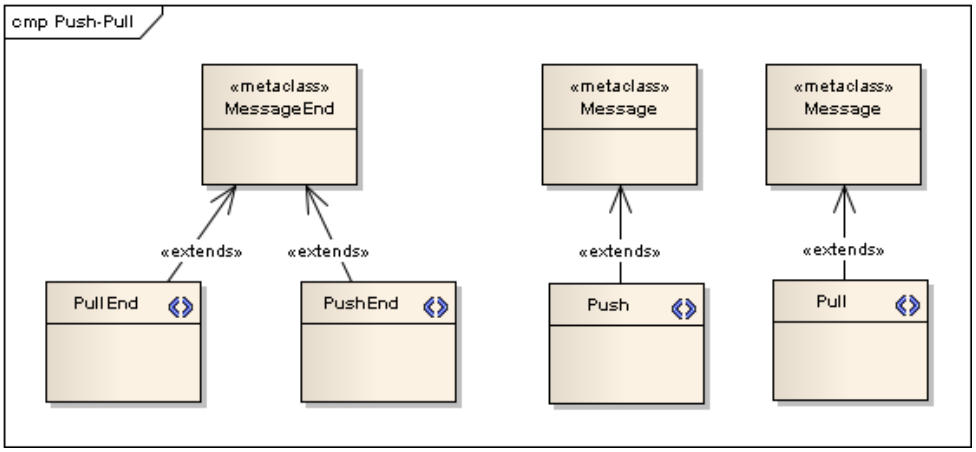

Figure 4.2: *UML Stereotypes For Modeling the Push-Pull Structure*

<<Push>>: A stereotype that extends the 'Message' metaclass and attaches to message ends that connect adjacent objects.

– A Push message has only two ends

$inv : self.baseMessage- > size() = 2$

– A Push message should be represented by a directed Message only

$inv : self.baseMessage.type.MessageEnd- > select(Message = Core :: MessageKind :: directed).class- > any(true)$

– The following constraint specifies the presence of interaction link between connected elements

$inv : self.enclosingInteraction- > select($
$oclAsKindOf(Message)- > exists(I : Interaction|I.PushEnd)$

<<Pull>>: A stereotype that extends the 'Message' metaclass and owns Message Ends that connect adjacent objects.

– A Pull message has only two ends

$inv : self.baseMessage.end- > size() = 2$

– A Pull message should be represented by a directed Message only

$inv : self.baseMessage.type.MessageEnd- > select(Message = Core :: MessageKind :: directed).class- > any(true)$

– The interaction contains the message ends owned by the adjacent objects

$inv : self.enclosingInteraction- >$
$select(oclAsKindOf(Message)- > exists(I : Interaction|I.PullEnd)$
$implies select(oclAsKindOf(Message)- > exists(I : Interaction|I.PushEnd)$

<<PullEnd>>: A stereotype that extends the MessageEnd metaclass and contains a number of operations that serve the purpose of Pull operations between connected elements.

$inv : self.baseMessageEnd- > forAll(i : Core :: MessageEnd|$
$PullEnd.baseMessageEnd- > exists(j|j = i)$

<<PushEnd>>: A stereotype that extends the MessageEnd metaclass and contains a number of operations that serve the purpose of Push operations between connected elements.

$inv : self.baseMessageEnd- > forAll(i : Core :: MessageEnd|$
$PushtEnd.baseMessageEnd- > exists(j|j = i)$

4.4.2 More Architectural Primitives

Due to space restrictions, we do not go into the detailed definition for the rest of the architectural primitives discovered in this work. Instead, we present a shortened modeling solution.

Callback

Textual Description: In a callback interaction between objects, an object B invokes an operation on object A, where object B keeps a reference to object A. Usually the callback function is invoked when a run-time event happens.

Known Uses in Patterns: MODEL-VIEW-CONTROLLER [5], OBSERVER [5], PUBLISH-SUBSCRIBE [5]

Modeling Issues: A major problem in modeling these patterns in UML is that even though callback is an active participant in the patterns, it can not be semantically represented in the interaction diagrams. A UML interaction diagram can depict the presence of a callback structure but it cannot be distinctively identified. It is hard to distinguish between many operations taking place between objects and the callback-specific operations.

Modeling Solution: To capture the semantics of callback primitive properly in UML, we use the following stereotypes: <<Callback>>, <<EventEnd>>, and <<CallbackEnd>>. The <<Callback>> extends the Message metaclass while the <<EventEnd>> and <<CallbackEnd>> extend the MessageEnd metaclass. A callback invocation is always preceded by an event occurrence and the callee object must have subscribed itself to the caller object beforehand. In this case, the kind of message communication must be of signal type [64] where the EventOccurence takes place at the sender object (EventEnd) while the EventExecution takes place at the receiver end (CallbackEnd).

Forward-Request

Textual Description: Forward-Request primitives are used to depict the presence of a request forwarding mechanism. Forward-Request messages decouple the underlying system from the external objects.

Known Uses in Patterns: PEERS [59], BROKER [5], CLIENT-SERVER[15], FORWARD-RECEIVER [59], MARSHALLER [59]

Modeling Issues: A Forward-Request typically differs from simple function calls, return calls, and other forms of communications among objects. The Forwarder object decouples the underlying system implementation from external function calls and converts incoming data into matching data format without introducing further dependencies. Moreover, in certain cases, the forwarder objects can receive return values that are forwarded to the source objects. However, UML elements cannot structurally express the presence of Forward-Request operations in software design.

Modeling Solution: To capture the semantics of Forward-Request properly in UML, we propose the following new stereotypes: <<Forward-Request>>, <<ForwardEnd>>, and <<ReceiverEnd>>. The <<Forward-Request>> extends the Message class and uses the <<ForwardEnd>> and <<ReceiverEnd>> to connect the adjacent objects. Both the <<ForwardEnd>> and <<ReceiverEnd>> extend the MessageEnd metaclass and are owned by the forwarder and receiver objects respectively. To execute an operation,, the <<ForwardEnd>> invokes the sendMessage operation, which is intercepted by the receiver object using the <<ReceiverEnd>>.

Command

Textual Description: Calling a method in the target object typically involves invoking a specific method or procedure in the target object. The invocation operation is usually carried out on the occurrence of a specific event.

Known Uses in Patterns: MODEL-VIEW-CONTROLLER [5], PRESENTATION-ABSTRACTION-CONTROL [59], LAYERS [5]

Modeling Issues: A command typically differs from data, events, and other forms of communications among objects. However, UML elements cannot structurally distinguish the presence of command operations in software design.

Modeling Solution: To capture the semantics of Command primitive properly in UML, we propose two new stereotypes: <<Command>>, and <<CommandEnd>>. The <<Command>> extends the Message class and uses the <<CommandEnd>> to invoke command on the target object when a specific event occurs. The <<CommandEnd>> extends the MessageEnd metaclass and is owned by the command invocation object.

Asynchronous Message

Textual Description: In an asynchronous communication, the message sender continues with its operation without waiting for any reply from the message receiver.

Known Uses in Patterns: PIPE-FILTER [5], CLIENT-SERVER [59], BROKER [5]

Modeling Issues: The patterns listed above often use Asynchronous messaging. UML supports the invocation of asynchronous messages when a specific event occurs. However, it does not enforce any constraints in distinctively recognizing the asynchronous operations. Various architectural patterns use degrees of asynchrony in their operations. In the most common form of asynchronous communication, the sender's data is buffered in queues without waiting for the recipient to pick the data. The current UML collaboration diagrams support the Asynchronous messaging; however, there are two major issues:

- Even though the UML diagrams have a support for Asynchronous messaging, they do not differentiate between the return values from the target objects. It is an ambiguous 'hint' to determine whether the return value is merely a notification event about the receipt of message or the actually processed data.
- Asynchronous messages are often buffered in queues until the target object notifies about its availability using events, often much later in the time. Such a structure cannot be un-ambiguously determined in UML interaction diagrams where a number of operations among objects are taking place at the same time.

Modeling Solution: We use the <<AsynchMessage>> stereotype along with the existing UML interaction diagram functions for modeling the asynchronous communication among the objects. The <<AsynchMessage>> extends the Message metaclass and uses the existing MessageSend and MessageReceive operations to guarantee that the invocation flag is active whenever an operation is invoked. We further constrain the Asynchronous communication to ensure that the method that invoked the operation is not bound to receive the results and only a notification event can inform the receipt of message.

Synchronous Message

Textual Description: In a synchronous communication, the sender waits till the receiver finishes the activated operation.

Known Uses in Patterns: PIPE-FILTER [5], CLIENT-SERVER [59], BROKER [5]

Modeling Issues: The patterns listed above often use Synchronous messaging. UML denotes a synchronous message with a solid arrowhead. We specify additional constraints on UML synchronous messages to provide a clear depiction of synchronous message.

Modeling Solution: We add a simple extension to the UML metamodel by proposing the <<SynchMessage>> stereotype for modeling the synchronous communication between objects. The <<SynchMessage>> extends the Message metaclass and uses the existing UML synchmessage operations to ensure that: a) a synchronous message is always represented with a directed association; b) an end-to-end connection is established with the target object, which owns the EventEnd and returns a flag each time a data processing is completed; and c) a return operation is mandatory for the synchronous communication to update the status of the operation that invoked the synchronous communication.

Call-Slave

Textual Description: The objects called slaves provide sub-services on behalf of a master object. The master also keeps reference to all the slave components.

Known Uses in Patterns: MASTER-SLAVE, PRESENTATION-ABSTRACTION-CONTROLLER [59], WHOLE-PART [59]

Modeling Issues: The call-slave structure is a key participant in modeling patterns when a task is delegated to a number of sub-objects. In such a case, the dependent objects work as slaves and usually do not invoke any operations on the surrounding elements. UML interaction diagrams can depict such a structure but cannot express the semantics in the diagrams.

Modeling Solution: We propose the following stereotypes to model the Call-Slave primitive: <<CallSlave>>, <<Slave>>, and <<Master>>. The <<CallSlave>> extends the Message metaclass and provides a selfMessage operation to invoke operations that further call upon slave objects. Both the <<Slave>> and <<Master >> represent the objects with further constraints such that only the <<Master>> object can access the <<Slave>> objects.

4.5 Modeling Architectural Patterns Using Primitives

In this section, we use the primitives described in the previous section to model known variants of three selected architectural patterns: Pipe-Filter, Model-View-Controller (MVC) and Client-Server. As aforementioned in the introduction, primitives capture only part of the semantics of the patterns, since there are semantics specific to individual patterns and not recurring in several patterns. Therefore, in order to complete the behavioral modeling of patterns, we need to find the missing pattern semantics and express them through a stereotyping scheme. Due to space limitation, we only provide detailed OCL constraints for the Pipe-Filter, while we omit the OCL code for the MVC and Client-Server.

4.5.1 Pipe-Filter

The Pipe-Filter pattern consists of a chain of data processing filters, which are connected through pipes. The output of one filter is passed through pipes to the adjacent filter. The elements in the Pipe-Filter pattern can vary in the functions they perform. For instance, pipes can buffer data, form feedback loops or fork/join structures, filters can be active or passive etc. Each such function can be described with a specific scenario to depict the behavior of the pattern. The

primitives discovered so far address many such variations. We select the Push, Pull, and Synchronous Message primitives from the existing pool of primitives. The rationale behind the selection of these primitives is as follows:

- The Push and Pull primitives are used to express the pipes that transmit streams of data between filters.
- Data is sent from one filter to the next filter in the chain using synchronous operations.

Missing Pattern Semantics: Despite the reusability support offered by the selected primitives, the Pipe-Filter pattern semantics cannot be fully expressed in design because the feedback, pipe, and filter structure are still missing. We apply the Feedback stereotype on the Push primitive to capture the presence of feedback loop in the Pipe-Filter pattern. Such a structure represents data being pushed from one filter object to another filter object using the feedback loop. The original Push primitive, as described in section 4, extends the UML metaclasses of Message and MessageEnd. The feedback stereotype further specializes the Push primitive by stereotyping it as Feedback without introducing new constraints.

<<Feedback>>: A stereotype that is applied on the Push primitive for expressing the Feedback operation in the Pipe-Filter pattern. The semantics of a feedback operation are similar to Push and Pull data streams operation.

The second stereotype named 'Filter' that we use from the existing vocabulary of design elements is defined as follows:

<<Filter>>: A stereotype that extends the Object metaclass of UML and owns message ends.

– A Filter object owns the MessageEnds of the associated pipes such that within an interaction, it owns the receiver end of source pipe and the sender end of next pipe in the chain

$$inv : self.enclosingInteraction- > select(oclAsKindOf(Object)- > exists(I : Interaction| I.MessageOut) implies self.enclosingInteraction- > select(oclAsKindOf(Object)- > exists(I : Interaction|I.MessageIn)$$

<<MessageOut>> A stereotype that extends the MessageEnd class and owned by the filter objects

$$inv : self.enclosingInteraction- > select(oclAsKindOf(Message)- > exists(I : Interaction|I.MessageOut)$$

<<MessageIn>> A stereotype that extends the MessageEnd class and owned by the filter objects

$$inv : self.enclosingInteraction- > select(oclAsKindOf(Message)- > exists(I : Interaction|I.MessageIn)$$

The fifth stereotype that we use from the existing vocabulary of design elements is the 'Pipe' that is defined as follows:

<<Pipe>>: A stereotype that extends the Message metaclass of UML and attaches the MeesageEnd of source object to the MessageEnd of the target object.

As shown in the figure above, the first filter object pulls data from the source object, and after processing pushes this data to the next filter in the chain. The second filter sends data back to the first filter using feedback pipe for further processing, and sends the final processed data to the sink.

4.5.2 Model-View-Controller

The behavior of MVC pattern relies on the functions performed by the following elements: Model, View, and Controller. The Model provides the functional core of the application and

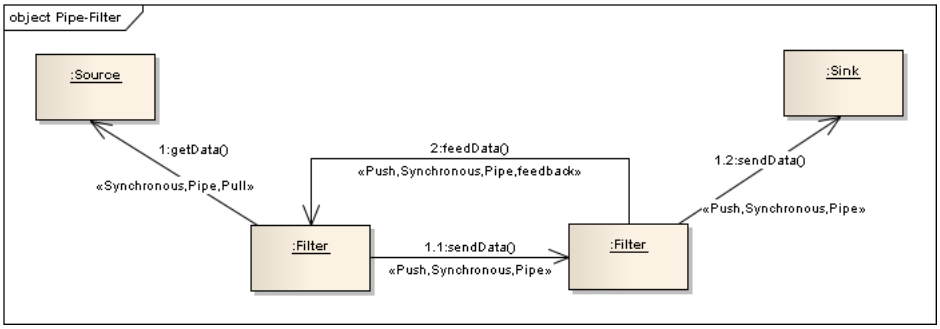

Figure 4.3: *Modeling Pipe-Filter Pattern Using Primitives and Design Elements*

notifies views about data changes. Views retrieve information from the model and display it to the user. Controllers translate events into requests to perform operations on the view and model elements. As a first step, we map the MVC pattern to the list of available primitives. We select the callback and command primitives for modeling the MVC pattern. The rationale behind the selection of these primitives is as follows:

- The view subscribes to the model to be called back when some data change occurs.
- Controller issues a command request on the model and view objects when some event occurs.

Missing Pattern Semantics: However, not every aspect of the MVC pattern can be modeled using the existing set of primitives. For instance, the Model, View, and Controller objects are not mapped to any primitives discovered so far. Keeping in view the general nature of these objects and their mandatory use in modeling different variants of the MVC pattern, we include the <<Model>>, <<View>> and <<Controller>> stereotypes in the existing vocabulary of pattern elements, as described below.

<<Model>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Controller and View objects.

<<Controller>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Model and View objects.

<<View>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Model and Controller objects.

4.5.3 Client-Server

In a typical Client-Server pattern variant, the server offers operations that are accessed by the clients and even clients can perform domain-specific operations at their own. Usually a broker pattern is used to establish connections between client and server. The client sends request to the broker asking to fulfill a specific task. The broker in response looks for the appropriate server and assigns the task to the server. The server provides the functional core of the application and uses the broker to send information back to the clients.

As a first step, we map the Client-Server pattern to the list of available primitives. We select the forward-request, asynchronous, and command primitives for modeling the Client-Server pattern. The rationale behind the selection of these primitives is as follows:

- The Server issues a command request to the clients when some event occurs.

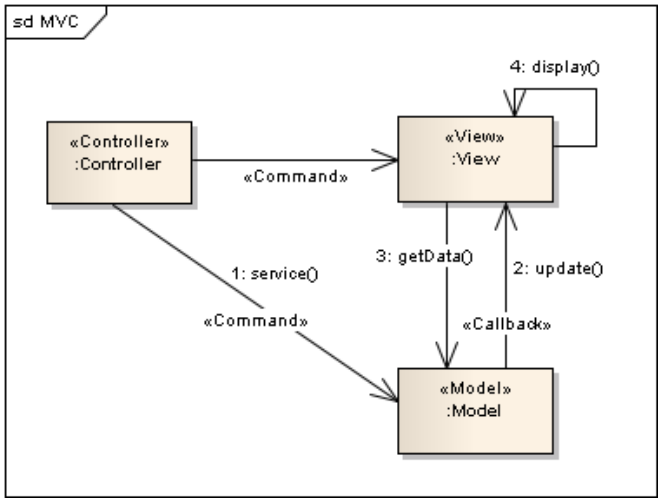

Figure 4.4: *Modeling the MVC Pattern Using Primitives and Design Elements*

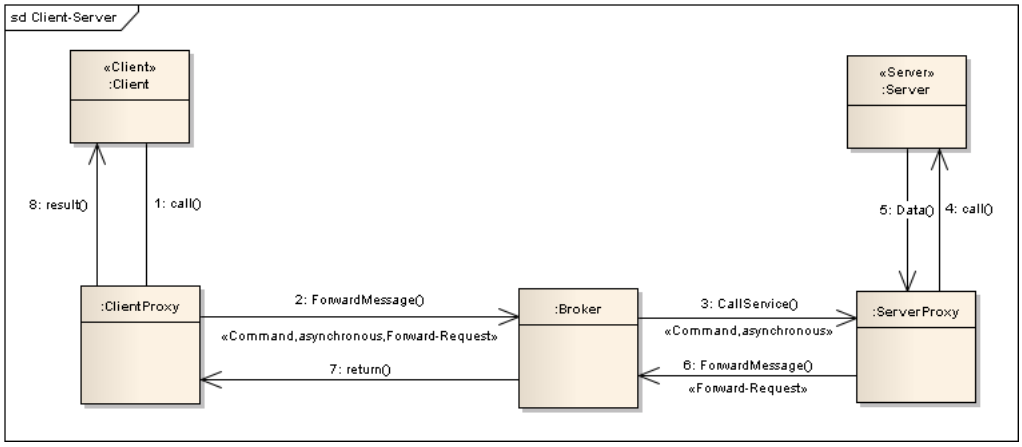

Figure 4.5: *Modeling the Client-Server Pattern Using Primitives and Design Elements*

- The Client and Server side proxies synchronously forward requests to other objects.

Modeling Pattern Semantics: However, not every aspect of the Client-Server pattern can be modeled using the existing set of primitives. For instance, the Client, and the Server objects are not mapped to any primitives discovered so far. Keeping in view the general nature of these objects, we provide reusability support by making these two pattern participants available in the existing vocabulary of design elements. <<Client>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Server and mediator objects. <<Server>>: A stereotype that extends the Object metaclass of UML and owns message ends for interaction with Client, surrounding objects, and mediator objects.

4.6 Related Work

The work described in this chapter is based on our previous work [15] where we present an initial set of primitives for modeling architectural patterns in the component-connector view. However, the idea to use primitives for software design is not novel and has been applied in dif-

ferent software engineering disciplines [67]. The novelty of our work lies in the use of primitives for systematically modeling the behavior of architectural patterns.

Using different approaches, other researchers have been working actively on the systematic modeling of architectural patterns. Garlan et. al. [35] propose an object model for representing architectural designs. They characterize architectural patterns as a specialization of object models. However, each such specialization is built as an independent environment, where each specialization is developed from scratch using basic architectural elements. Our approach significantly differs as our focus lays on reusing primitives and pattern participants, which are defined as specializations of UML elements.

Werner et. al. [68] uses message sequence charts to propose a language that is capable enough to fully express the behavioral specification of systems using use cases and scenarios. Their work focuses on the execution of scenarios when different kinds of events occur for message calls of type e.g. asynchronous message, synchronous message. In our approach, we also use messages as a base for interaction but our focus revolves around modeling patterns where we use primitives and pattern participants' definitions as reusable abstractions.

4.7 Conclusions

The combination of architectural primitives and the vocabulary of design elements offers a systematic way to model patterns' behavior in system design: the primitives and the design elements are reusable architectural abstractions in the form of extended UML elements; the semantics of the primitives and subsequently of the patterns can be validated by checking the OCL constraints; the patterns can be manually or automatically detected in the system design. In this chapter, we have extended our existing pool of primitives with the discovery of seven more primitives in the behavioral view. Moreover, with the help of some example patterns, we demonstrated the feasibility of our approach for modeling architectural patterns using primitives.

Parts of this chapter has been published/submitted as:

1. A. W. Kamal and P. Avgeriou – *"Modeling the Variability of Architectural Patterns"*, proceedings of the ACM Symposium on Applied Computing, Software Engineering track, pp. 2344-2351, March 22 - 26, 2010, Sierre, Switzerland.
2. A. W. Kamal, P. Avgeriou, and U. Zdun – *"Designing Software Architectures Using Architectural Patterns Variants: A Controlled Experiment,"* Submitted for review in the Journal of Software Practice and Experience.

Chapter 5

Modeling the variability of architectural patterns

Abstract

Architectural patterns provide solutions to recurring design problems in the context of a software architecture at hand. Most architectural patterns have numerous variations which can be used to realize a software architecture. Effectively specializing and applying patterns to software architectures remains a challenging task mostly because of the inherent pattern variability and because the architectural abstractions of modeling languages do not match pattern participants. In this chapter, we demonstrate the use of a pattern variant modeling approach for effectively modeling architectural patterns variants within software architectures. The approach is validated in the context of a controlled experiment for designing a software architecture.

5.1 Introduction

Architectural patterns provide solutions to recurring problems at the architecture design level. Architectural patterns are seldom applied 'as is' to solve a design problem. Often the solution specified by architectural patterns needs to be specialized in the context of a software architecture; this leads to several different variants of a pattern [5]. In essence, the solution specified by a pattern provides only guidelines to solve a design problem and leaves blank spaces that need to be filled in by software architects [5]. This requires specialization of a pattern's solution to best fulfill the design requirements at hand. For instance, Feedback loops can optionally be added to the solution of the Pipes and Filters architectural pattern where each such loop is a specialization to the pipe participant[1] of the Pipes and Filters pattern [5]. So far a long list of architectural patterns and pattern variants have been documented in the literature [4, 59], and this list is growing continuously. Each of these patterns has numerous documented and undocumented variations similar to the Feedback loop example in Pipes and Filters.

There are four approaches that have been used so far to express the solution specified by a pattern in system design:

- Architectural Description Languages (ADLs) [16] that have been traditionally used for describing software architecture.
- Unified Modeling Language (UML) [46] [49], which is a widely used generic modeling language for designing systems in different domains but is mostly used to design software.
- Formal approaches [43] that specify precise pattern solution to specific problems e.g. pattern-specific components and connectors.

[1]The term pattern participants, frequently used in this chapter, refers to the solution participants of architectural patterns. For example, the Pipe and the Filter are solution participants of the Pipes and Filters pattern [5].

- Informal box and line diagrams that provide little information about actual computation represented by boxes, and the nature of the interactions between them [42].

In spite of the benefits that these 4 approaches offer, there is not yet an established approach for effectively expressing and applying pattern variants in a system design. Current approaches for designing software architecture such as described above do not provide explicit support for modeling pattern variants [28]. UML offers only generic architectural elements (i.e., components and connectors) [64] while other modeling languages (e.g., ACME [38], Wright [53], Aesop [39]) provide support for modeling a limited set of architectural patterns but do not address the challenges of modeling different variants of patterns. Similarly, pattern formalization approaches [43, 69] do not support modeling of variants either, as they provide formalized solutions of patterns which may narrow the applicability of a pattern in expressing different system-specific solutions. Moreover, the architectural abstractions present in the modeling languages do not match the participants of pattern variants [28]. For instance, the pipe participant of the Pipes and Filters pattern does not match the Connector element present in UML [15]. Current approaches are either too generic or only provide support for specific design solutions which limits their applicability to effectively grasp the whole solution space covered by architectural patterns.

In our previous work [15, 30], we have identified a set of architectural primitives. These primitives offer reusable modeling abstractions as architectural building blocks that support modeling a number of architectural patterns. In order to express the complete solution of pattern variants, primitives are not enough; there are missing solution aspects of pattern variants that need to be addressed. For instance, Feedback Loops, Forks, and Joins are variant-specific participants of the Pipes and Filters pattern, which are not addressed by primitives. In this chapter, we devise a pattern variants modeling approach that is aimed at helping inexperienced architects with modeling several variants of patterns in an effective way. The approach uses architectural primitives in combination with variant-specific participants for modeling pattern variants. The approach has been empirically validated through a controlled experiment. The validation brings evidence that the use of the proposed approach a) helps to effectively model several variants of architectural patterns, and b) assists in system decomposition into components and connectors during the phase of software architecture design.

The remainder of this chapter is structured as follows: Section 5.2 compares our approach to the related work. In Section 5.3, we briefly introduce the notions of architectural patterns, pattern variants[2], and discuss related modeling languages. Section 5.4 documents the approach for modeling pattern variants, gives an overview of the architectural primitives discovered in our previous work, and provides a detailed mapping between the selected pattern variants and primitives. In Section 5.5 we apply our approach in the context of an example software architecture, designed using the approach presented in this chapter. Section 5.6 details the design of the experiment and in Section 5.7, we document the execution of the controlled experiment. Section 5.8 presents statistical results from the controlled experiment. Section 5.9 interprets qualitative data gathered after the experiment and discusses the possible threats to the validity of the results. Section 5.10 concludes this study.

5.2 Related work

The work described in this chapter is based on our previous work [15, 30] where we present a set of primitives for modeling architectural patterns. The idea to use primitive abstractions

[2]For the sake of simplicity, we shall use the term 'pattern variants' instead of 'architectural pattern variants' in this chapter.

and UML extensions for software design is not novel and has been applied in different software engineering disciplines [61]. The novelty of our work lies in the use of primitives for effectively modeling architectural patterns and patterns' variants, which has not been fully addressed before.

Using different approaches, few other researchers have been working actively on the systematic modeling of architectural patterns [35, 61, 62]. Garlan et al. [35] proposes an object model for representing architectural designs. They characterize architectural patterns as specialization of the object models where each such specialization is built as an independent environment to be applied in a specific project. They describe a system called Aesop for developing pattern-oriented architectural design environments. However, their work is focused on a well-known set of documented architectural patterns and in-depth understanding of the design environment is required to define new patterns. Our focus is on reusing primitives and pattern participants and only where required we extend the UML elements to capture missing pattern semantics.

Giesecke et al. [62] extend the UML metamodel by creating pattern-specific profiles. Their work maps the MidArch ADL to the UML metamodel for expressing patterns within software architectures. However, their approach does not address modeling several variants of a pattern either. Manual work is required to extend UML metaclasses for expressing new pattern variants. Our approach distinctively differs from their work as we focus on expressing a generalized list of pattern variants rather than individual patterns.

Snirc et al. [70] introduces UML extensions to support feature modeling. Feature modeling, similar to what we present in this chapter, is the activity of modeling the common and variable aspects of a system. Still, our work differs from feature modeling as features are mostly defined around concepts and not classes, objects, or individual pattern participants. For instance, a pattern can itself be considered as a feature within software architecture. Our work, as compared to feature modeling, is at a more detail level of abstraction where we focus on the solution participants of architectural patterns for defining several pattern variants.

Kim et al. [71] propose a simple approach based on defining pattern-specific UML metamodels by constraining the meta-models defined at UML's M2 level (see UML superstructure for detailed description of UML levels [64]).Using extensions to UML metaclasses, they define static pattern specification using class diagrams and interaction pattern specification using sequence diagrams. However, providing automated support for defining patterns in different views is non-trivial because of the extensive use of specialized pattern-specific notations especially for defining patterns. In the absence of a systematic approach to define different patterns, we found the notation used in their work require extensive design effort to model patterns.

Some work has been done to use UML to define and document patterns. The OMG [64] introduces 'collaborations' for modeling design patterns and Klaus [72] defines pattern-specific profiles. However, these approaches are focused on modeling design patterns and do not specifically address the issues related to variations in the pattern solutions. Though our work too uses UML extensions to define primitives, generic and specialized participants, the profiles defined in our work are specifically focused to model architectural pattern variants. Moreover, existing UML based approaches are focused on detail level architectural abstractions, i.e., classes, operations, and data types, while our work is focused on relatively high level architectural elements, i.e., components and connectors.

In addition to the UML-based pattern modeling approaches discussed above, several other extensions to UML meta-model are proposed for modeling patterns. Guennec et al. [73] and Mak et al. [74] use meta-level collaborations to present design patterns and specify some pattern properties as a set of constraints using Object Constraint Language (OCL). Dong et al. [75] define a UML profile for design patterns and also provide a web-service for visualizing patterns

Approach	Elements	Application	Scope	Contribution
Object models [35]	Classes and associations	Architecture design	Project specific	Characterization of architectural patterns as specialization of object models
Classification framework [61]	Software connectors	Architecture design	Component-based development	Composing architecture building blocks for complex software interaction
Pattern-specific UML profiles [71] [72]	Classes and associations	Architecture design	Project specific	Maps midArch ADL to a UML meta-model
Feature modeling [70]	Architecture design concepts	Architecture design	General	Modeling common and variable aspects of a system
UML extensions for modeling patterns [64]	UML classes and associations		General	UML collaborations for modeling design patterns
UML meta-models for modeling patterns [62]	UML static and interaction diagrams	Classes and objects	General	A meta-model to specify design patterns
Architectural specification methods for expressing patterns [43]		Classes, Relations, Actions	General	Temporal logic of actions

Table 5.1: *Overview and comparison of related work*

in UML diagrams. All these researchers discuss patterns in the context of UML and limit their application to UML, while our work discusses patterns independently from a particular modeling language such as UML.

Mehta et al. [61] focus on the fundamental building blocks of software interaction and the way these can be composed into more complex interactions. They present a classification framework with a taxonomy of software connectors and advocate the use of the taxonomy for modeling software architecture. However, the taxonomy lacks the information to model a variety of architectural patterns rather it is focused on the basic building blocks of component-based development. Our work focuses on effectively designing software architecture using primitives, generic and specialized pattern participants that provide architectural building blocks in context of pattern variants.

Mikkoenen [43] define patterns behavior using a specification method called DisCo, which is intended for specification and modeling of interactions at a high-level of abstraction. The formal basis of the method is in Temporal Logic of Actions[43]. They use classes, relations, and actions to express an example design pattern. Their work is focused on object-oriented modeling capabilities for developing system specification i.e. using class diagrams which does not match the abstractions of architectural patterns like Pipes and Filters and Layers.

Table 6.1 provides a comparison and overview of related work in the field of software architecture for modeling patterns.

5.3 Background - Architectural patterns, pattern variants and modeling languages

5.3.1 Architectural patterns and design patterns

Over the last decade, architectural patterns have increasingly become an integral part of the software design practice [8]. Architectural patterns provide solutions to recurring design prob-

lems for software architectures. For instance, often the patterns can be realized using a component and connector model [39, 40]. In addition to architectural patterns, there are also patterns documented in the literature that provide detailed design solutions which are more close the actual implementation of a software, such as object-oriented design patterns in [7]. In comparison to design patterns, architectural patterns pose more software modeling challenges mainly because of the variation in applying architectural patterns to different software architectures. Still, it is difficult to draw a clear boundary between these two types of patterns, and it largely depends on the way these patterns are perceived and used by software architects. Our work in this chapter is mainly concerned with architectural issues, thus focusing on architectural patterns, i.e., patterns dealing with system-wide components and connectors rather than implementation-specific details.

5.3.2 Architectural patterns variants

Architectural patterns are rarely modeled within a software architecture in their original form. The solution portrayed by an architectural pattern can be specialized in a number of ways to design a software architecture. This realization of architectural patterns for designing software architectures yields architectural patterns variants. For instance, Binary Filters specialize the Filter participant of the Pipes and Filters pattern by constraining Filters to attach exactly one input and one output port [5]. The existence of pattern variants in several different forms makes it difficult to effectively express patterns within a software architecture. Often extensive design effort is required to effectively model the participants of pattern variants, which is the focus of our work in this chapter.

5.3.3 Modeling languages for designing software architectures

Modeling languages have traditionally been used to model patterns in general and software design in specific. Software architects use the inherent as well as the extensible support of modeling languages to express architectural patterns within software architectures [15, 35, 46, 38, 32]. Many of these modeling languages focus on the use of generic components and connectors as architecture building blocks. For example, some modeling languages, such as ACME [38], Aesop [39], and Unicon [35] provide built-in support for pattern-specific architectural elements to model patterns. In particular, ACME supports templates that can be used as recurring patterns, Aesop allows pattern-specific use of vocabulary, and UniCon provides syntax and graphical icons support for a limited set of patterns. However, there is not yet a widely accepted approach for effectively modeling pattern variants within a software architecture. Unfortunately, the current software architecture design practices do not specifically address the problem of modeling several pattern variants.

5.4 An approach to model pattern variants within software architecture

The approach for modeling pattern variants, presented in this section, is based on grouping pattern participants into three categories, namely: architectural primitives, generic pattern participants and specialized pattern participants. We first briefly describe these terms, document the approach for modeling pattern variants, and finally provide a mapping between the pattern variants and primitives.

5.4.1 Architectural primitives, generic and specialized pattern participants

Architectural primitives: In our previous work [15, 29], we have listed a set of architectural primitives along with the mechanism to discover the primitives in architectural patterns. Architectural primitives, as recurring solution participants of architectural patterns, contribute to model several architectural patterns variants. Our original set of primitives is comprised of the primitives summarized in Table 5.2 (see also [15, 29]).

Architectural Primitives	Description
Callback	A Component B invokes an operation on Component A, where Component B keeps a reference to Component A in order to call back to Component A later in time.
Indirection	A component receiving invocations does not handle the invocations on its own, but instead redirects them to another target component.
Grouping	Grouping represents a Whole-Part structure where one or more components work as a Whole while other components are its parts.
Layering	Layering extends the Grouping primitive, and the participating components follow certain rules, such as the restriction not to bypass lower layer components.
Aggregation Cascade	A composite component consists of a number of subparts, and there is the constraint that Composite A can only aggregate components of Type B, B only C, etc.
Composition Cascade	A Composition Cascade extends Aggregation Cascade by the further constraint that a component can only be part of one composite at any time.
Shield	Shield components protect other components from direct access by the external client. The protected components can only be accessed through Shield.
Typing	The Typing primitive introduces the notions of a 'supertype' connector and a 'type' connector, which can be used to define custom typing models using associations.
Virtual Connector	Virtual connectors reflect indirect communication links among components for which at least one additional path exists from the source to the target component.
Push-Pull	Push, Pull, and Push-Pull structures are common abstractions in many software patterns. They occur when a target component receives a message on behalf of a source component (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull).
Virtual Callback	In many cases the callback between components does not exist directly, rather there exist mediator components between the source and the target components. For instance, in the MVC pattern, a model may call a view to update its data but this data may be rendered first by the mediator components before it is displayed to the end-user.
Adapter	This primitive converts the provided interface of a component into the interface the clients expect.
Passive Element	Consider an element is invoked by other elements to perform certain operations. Passive elements do not call operations on other elements.
Interceder	Sometimes certain objects in a set of objects cooperate with several other objects. Allowing direct link between such objects can overly complicate the communication and result in strong coupling between objects. To solve this problem, Interceder components are used.

Table 5.2: *Architectural Primitives Description*

Pattern participants: In addition to primitives we consider two more concepts that allow to model the rest of the semantics in pattern variants: generic pattern participants and specialized pattern participants.

- *Generic Pattern Participants:* The solution of architectural patterns, as documented in the literature, is often generic in nature. To address this, the notion of generic participants is introduced, that are customizable to the requirements of a specific system. The term 'generic pattern participants' refers to the solution participants of a pattern in their original form as documented in the literature. For example, a generic Filter represents a data processing unit without going into the detail of input/output and communication protocols. Generic participants are customizable w.r.t. the specific needs of a system which makes them reusable for modeling several different variants.

- *Specialized Pattern Participants:* The specialized participants are variant-specific specializations of generic participants. The specialized participants' describe the solution

of pattern variants and relationships to solve specific design problems, e.g., the Document and the Page Controller are specialized participants within the Document-View and Page Controller variants of the MVC pattern respectively [5]. Pattern variants can be modeled by integrating one or more specialized pattern participants. This allows a designer to model unique pattern variants by using different combinations of specialized pattern participants.

The next sub-section documents the mapping of pattern variants to a set of architectural primitives.

5.4.2 Mapping primitives to pattern variants

Depending on the requirements of a software project, different pattern variants may convey different design solutions. In essence, the selection of primitives for modeling pattern variants lies with an architect who selects the primitives that best fit to solve a design problem. For instance, in the Pipes and Filters pattern, a filterA may push data to filterB, pull data from filterB, or even both the Push and Pull structures can exist at the same time. Table 5.3, as an example, provides a mapping of the selected pattern variants to architectural primitives.

Arch. Patterns	Pattern Variants / Primitives	Callback	Indirection	Grouping	Layering	Aggregation Cascade	Composition Cascade	Shield	Typing	Virtual Connector	Push-Pull	Virtual Callback	Adapter	Passive Element	Interceder
MVC	Model-View-Presenter	*	*								*	*			
	Passive Model MVC		*								*			*	
	Document View	*	*								*	*			
	Page Controller	*	*								*	*			
	Front Controller	*	*								*	*			
	Model-GUI-Mediator														
Layers	Relaxed Layers		*		*					*	*				
	Adapter Layer		*		*			*			*		*		*
	Layering Through Inheritance		*		*			*			*		*		*
	Strict Layering		*												
	Indirection Layer		*												
Broker	Adapter Broker		*	*				*					*		*
	Callback Broker	*	*	*				*							*
	Direct Communication Broker		*	*						*					
	Asynchronous Broker	*	*	*				*				*			*
	Client-Dispatcher-Service	*	*				*								
	Message Broker	*	*				*								
Pipes and Filters	Tee and Join Pipes and Filters										*		*	*	
	Feedback Pipes and Filters	*									*		*	*	
	Forks Pipes and Filters	*									*		*	*	
	Binary Pipes and Filters										*			*	
Proxy	Remote Proxy	*	*					*		*		*			
	Protection Proxy	*	*					*							
	Cache Proxy	*	*					*				*		*	*
	Synchronization Proxy	*	*					*				*			
	Counting Proxy	*	*					*		*		*			*
	Virtual Proxy	*	*									*			
	Firewall Proxy	*	*					*							

Table 5.3: *Pattern Variants to Primitives mapping*

The detailed discussion about the discovery of each primitive in the related patterns solutions is already documented in [15]. In essence, the mapping from patterns to primitives is not fixed because different variants of a selected pattern can be modeled using different combinations of primitives. For instance, the MVC pattern, as documented in [5], uses the *Callback*,

Push-Pull and *Virtual Callback* primitives. However, two different variants of the same pattern namely the Adaptive Model MVC [40] and Passive View MVC [5] benefit from the *Adapter* and *Passive Element* primitives respectively. This mapping of MVC pattern variants to different primitives not only strengthens the approach to express pattern variants using a set of primitives but provides a wider reusability and model checking support to design a software architecture using pattern variants. Table 5.3 provides mappings of few known pattern variants to architectural primitives.

5.4.3 An approach to model architectural pattern variants

The architectural primitives, generic and specialized participants, described in the previous two subsection, can be used to model pattern variants. We suggest a few simple steps to reach this goal.

The pre-requisite for modeling pattern variants with primitives is the selection of pattern variant(s) that best fit to a design problem at hand. To model pattern variants within a software architecture primitives that participate in modeling a selected pattern variant are selected using Table 5.3. If the primitives alone are not sufficient to express the complete solution of a pattern variant, it is checked if the generic and specialized participants of the selected pattern variant are present. Next, if specialized participants are not present to express the pattern variant then the generic pattern participants are specialized to express the selected pattern variant. To demonstrate these steps for modeling pattern variants, an example process is shown in Figure 5.1. It shows the steps of defining generic and specialized pattern participants. Two detailed example for these steps in modeling pattern variants with primitives, generic and specialized participants are given in section 5.5 and Appendix E.

5.5 Modeling architectural patterns variants: An example software architecture design

In this section, we design part of an example software architecture to demonstrate the use of the pattern variant modeling approach documented in the previous section. The basic functionality of the IS2000 software [2] is to acquire images as raw data and convert them into sensor readings and images suitable for viewing. The software has a set of acquisition procedures aimed at customized acquisition of the images. The product requirements are expected to change during the development and future life span of the software. The current design of the software does not employ pattern modeling. However, a documentation study of the software design revealed variants of three prominent architectural patterns, namely Pipes and Filters, Layers, and Model-View-Controller.

We use the approach documented in Section 5.4 for modeling variants of the aforementioned three architectural patterns within the IS2000 software architecture. We provide only an excerpt of the design i.e. we cover only the Component-Connector view and leave certain sub-systems un-touched. However, the primitives we discovered in other views can be used to design the same software architecture in different architectural views such as the process flow view [76], interaction diagrams [30] etc.

5.5.1 Expressing the pipes and filters pattern variant

One prominent architectural pattern used in the IS2000 software architecture is the Forked Pipes and Filters pattern variant providing a chain of image processing functions. The chain

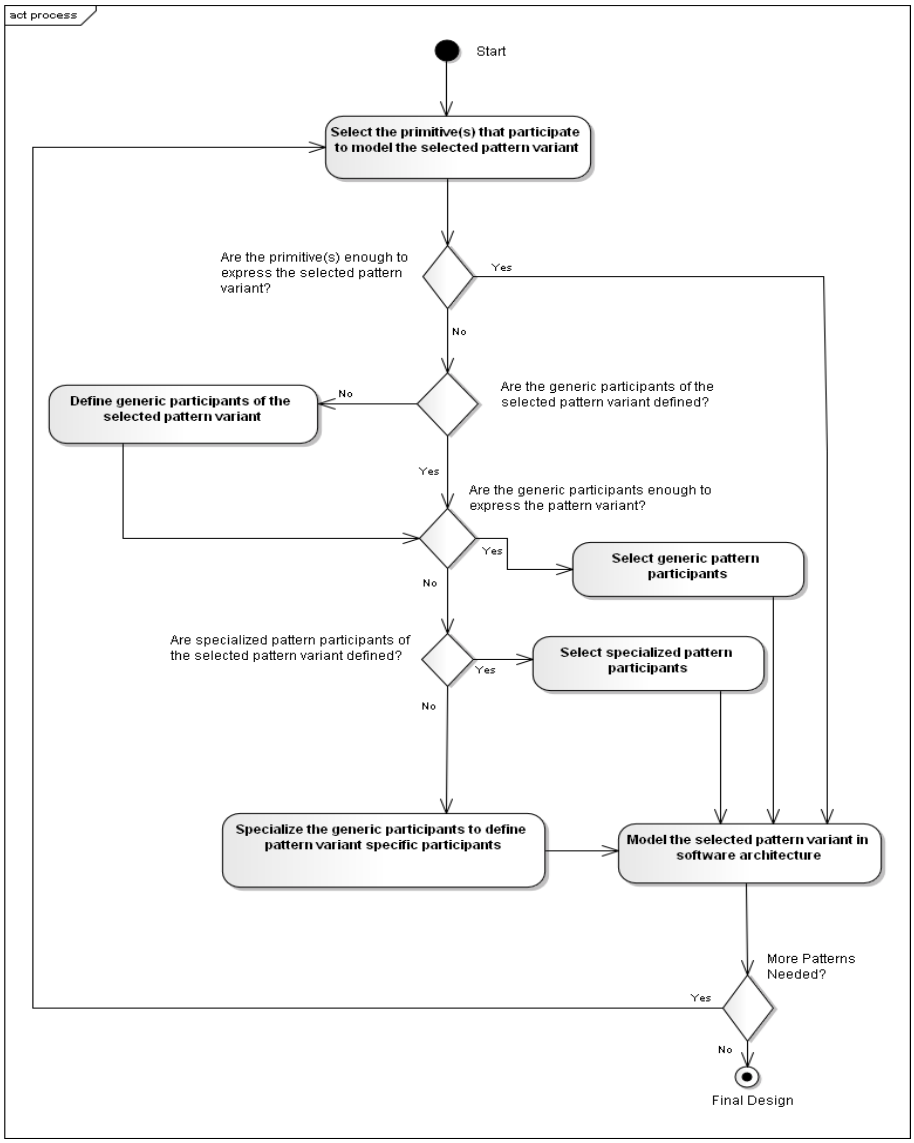

Figure 5.1: *The Process to Model Pattern Variants within software Architecture*

receives input as raw data and after performing a number of processing tasks produces refined data to the user and database. As shown in Figure 5.2, each filter in the Imaging component acts as a data processor by receiving input at one end and forwarding processed data to the next filter in the chain. The Acquisition component controls the data processing, the Imaging component receives and processes the raw data, and the Exporting component sends data to other systems.

The Adapter and Push-Pull primitives are selected for modeling the Pipes and Filers pattern variant in accordance to the information documented in Table 5.3. The rationale for selecting these primitives is as follows:

- In the Acquisition component, the ProbeControl, AcquisitionManagement, and Acquire components control the processing of data and convert the incoming data in a suitable form to be sent to the Framer and Imager components that is expressed using the *Adapter* primitive.

- In the Imaging and Exporting components where the actual data processing takes place; the flow of data is expressed using the *Push* primitive.

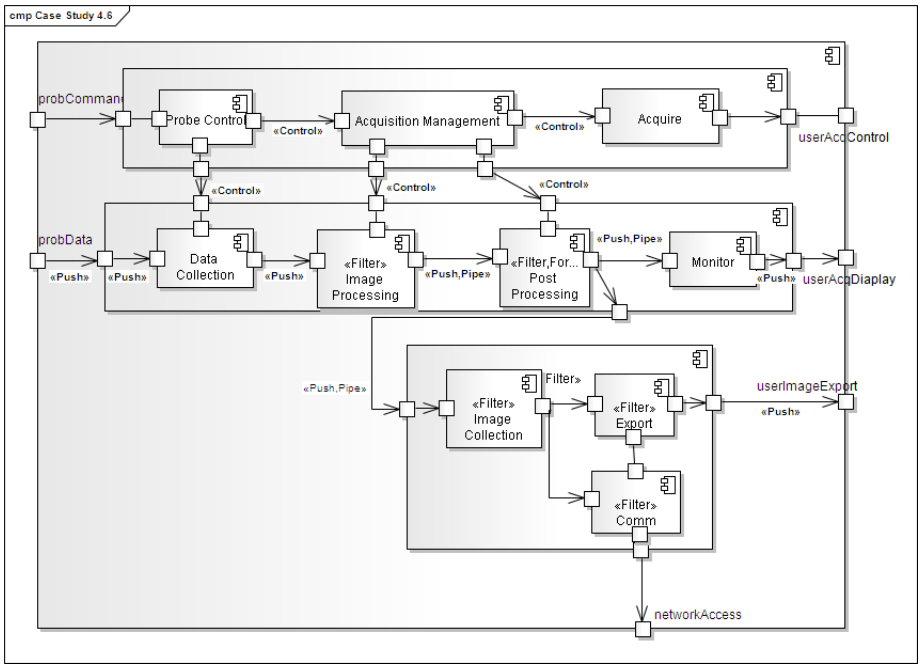

Figure 5.2: *Pipes and Filters Pattern Variant Expressed in IS2000 software*

The PostProcessing component pushes data to both the Monitor and the Exporting component, forming a *Fork* which is a variation from the documented Pipes and Filters pattern. The Fork participant specializes the Filter participant of the Pipes and Filters pattern. As a next step, the generic participants of the Pipes and Filters pattern variant are defined i.e. the Filter and Pipe alongwith the Input and Output ports. Next, the generic participant Filter is specialized to define the Fork participant, which is a variant-specific participant of the Pipes and Filters pattern.

After defining the generic and specialized participants of the Pipes and Filters pattern, the data processing architecture of the IS2000 system is designed as followed:

- The *Filter* participant is applied to the sub-components of the Imaging and Exporting components.
- The *Pipe* participant is used to express the flow of the image data between the filters.
- The *Fork* participant is applied to the Post Processing component to forward the input to next two filters in the chain as shown in Figure 5.3.

5.5.2 Expressing the layers pattern variant

Layers is the second most prominent architectural pattern used in the IS2000 software for the logical grouping of components. The IS2000 software architecture allows components in individual layers to bypass adjacent layers to send and receive information. Such a structure is called the 'Relaxed Layers' which is a variant of the Layers pattern as documented in [5].

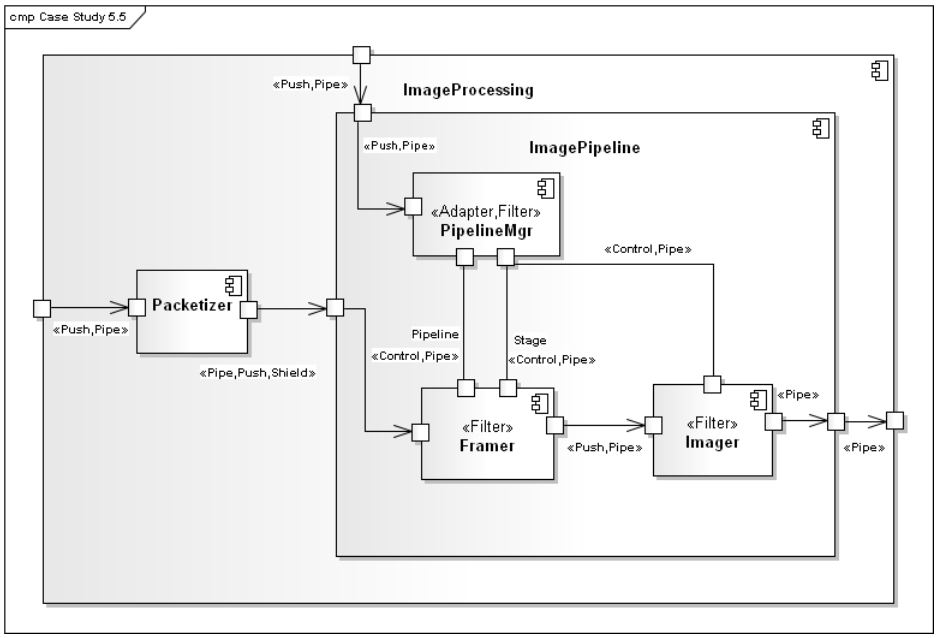

Figure 5.3: *Internal Structure of the ImageProcessing Component*

As per the first step for modeling the Relaxed Layers pattern variant, the Layering, Virtual Connector, Interceder, and Shield primitives are selected. The rationale for the selection of these primitives is as follows:

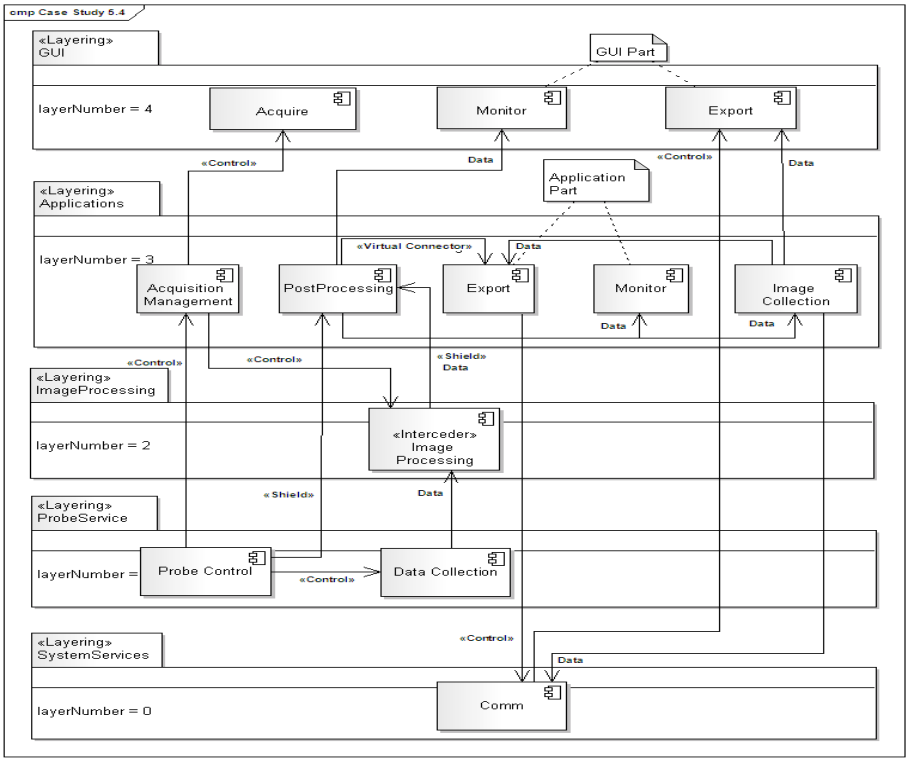

Figure 5.4: *Layers Pattern Variant expressed using architectural primitives*

- The Layering primitive is used to express the Layering structure within IS2000 software architecture.
- The PostProcessing component sends data to both the Monitor and Image Collection components, which forward this data to the Exporting component; forming a virtual connection between the PostProcesing and the Export component. This connection is expressed using the *Virtual Connector* primitive.
- The ImageProcessing component mediates the communication between the DataCollection and the PostProcessing component; expressed using the *Interceder* primitive.
- The Export, Monitor and Image Collection components can only be accessed through the PostProcessing component. That is, the connection from the ImageProcessing to the PostProcessing expressed using the *Shield* primitive.
- We relax the *Layering* primitive constraints to let layers freely bypass each other forming a 'Relaxed Layers' pattern variant.

The layering structure is effectively expressed using the existing set of primitives without a need to use/define specialized pattern participants for the Layers pattern. Figure 5.4 shows the Relaxed Layers pattern variant expressed within IS2000 software architecture.

5.5.3 Expressing the MVC pattern variant

In the example IS2000 software, the GUI module is responsible for defining and managing the user display and handle user events, whereas the core functionality that defines necessary action when an event takes place is handled by the application module. The GUI module can accept input from the mouse, keyboard, or the screen menus to which the application module set up the acquisition parameters, forward messages, or report the status of acquisition to the GUI. Thus, the display and event handling are handled by the GUI module while the application logic resides in the application module. This kind of structure is known as the Document-View architecture [5], which is a variant of the MVC pattern. The Document component corresponds to the Model in MVC while the View component of the Document-View merges the Control and View components.

As a first step for modeling the Document-View pattern variant, we select the *Callback*, and *Push* primitives from the existing set of primitives. The rationale for selecting these primitives is as follows:

- The Callback primitive is used to report status of acquisition back to GUI module.
- The Push primitive is used to send data to Monitor and Image Collection components.

Next, the generic participants of the MVC pattern are defined i.e. the Model, View and Controller. The Model, View, and Controller are generic participants of the MVC pattern that need further specialization to express the Document-View pattern variant. The Control and View participants are specialized by merging into a single component resulting into specialized View participant next.

After defining the generic and specialized participants of the MVC pattern, the Document-View variant of the MVC pattern is modeled as follows:

- The applications part of the IS2000 architecture is expressed using the Document participant while the GUI part is designed by applying the View participant.

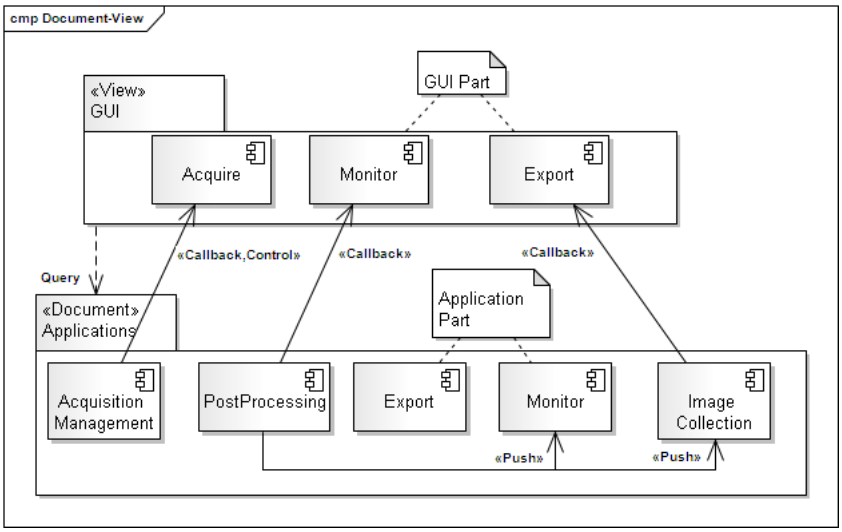

Figure 5.5: *Document View variant expressed using primitives and specialized pattern participants*

- The callback and notification links between the View and Document components are expressed using the callback and control primitives accordingly while the Push primitive is applied to send data from the PostProcessing component to the Monitor and Image Collection components as shown in Figure 5.5.

In the following sections, we present results from a controlled experiment for designing a software architecture using the proposed approach.

5.6 Experiment Design

The approach presented in the previous section is aimed at effectively expressing several variants of patterns for designing software architectures. To validate the use of primitives, generic and specialized pattern participants for modeling pattern variants, we performed a controlled experiment. In this experiment, two groups of post-graduate students were provided with the same information for designing software architectures. In addition, one group was instructed to use the pattern variant modeling approach. The experiment was designed according to the guidelines for conducting empirical research in software engineering [21] and the results were documented according to the reporting guidelines for controlled experiments in software engineering [27].

In the following sub-sections, the design of the experiment, the hypotheses, involved subjects, variables, and the analysis of the data collected from the experiment are presented. With the final data gathered from the outcome of the experiment, we evaluate how the use of architectural primitives, generic and specialized participants can help architects for effectively modeling pattern variants within software architectures.

5.6.1 Research Question and Hypotheses

To analyze the use of the pattern variant modeling approach, we present the research question and construct null hypotheses (H0i) and alternate hypotheses (H1i) as explained below.

Research Question: *Does the use of primitives, generic, and specialized participants help to effectively design a software architecture that includes pattern variants?*

Name of the variable	Class/Entity	Type of attribute (internal/external)	Scale	Measurement Unit	Range	Counting Rule
Architecture design experience	Software Architecture	External	Ordinal	Year	Above 5, 3 to 5, 1 to 3, No experience	Pre-experiment feedback
Experience of modeling patterns	Architectural Patterns	External	Ordinal	Expertise	Very Good, Good, Average, Little or no experience	Pre-experiment feedback
Understanding of candidate patterns	Architectural Patterns	External	Ordinal	Patterns count	More than 10, 8 to 10, 4 to 7, 0 to 3	Post-experiment feedback

Table 5.4: *List of independent variables*

The modeling of architectural patterns covers several aspects of software architecture design [77]. We have selected two such aspects that are evaluated according to the following hypotheses:

Null Hypotheses:

1. *H01:* The use of primitives, generic, and specialized participants does not help software architects to effectively express the solution specified by several pattern variants.
2. *H02:* The use of primitives, generic, and specialized participants does not help software architects to more effectively partition a software architecture into components and subcomponents, and assign responsibilities as compared to partitioning a software architecture without the use of such an approach.

Alternate Hypotheses:

1. *H01:* The use of primitives, generic, and specialized participants helps software architects to effectively express the solution specified by several pattern variants.
2. *H02:* The use of primitives, generic, and specialized participants helps software architects to more effectively partition a software architecture into components and sub-components, and assign responsibilities as compared to partitioning a software architecture without the use of such an approach.

5.6.2 Variables

We use independent variables to measure their influence on the final results. For instance, a participant having a very good understanding of architectural patterns may significantly influence the results gathered from a group.

Independent Variables: We considered three independent variables prior to conducting the experiment as listed in Table 6.3. All three independent variables namely architecture design experience, experience of modeling patterns, and understanding of candidate architectural patterns are measured according to ordinal scale as per the measurement scales documented in [78].

Dependent Variables: The experiment used two dependent variables, as shown in Table 6.4 for analyzing the software architectures. The dependent variables correspond to the two hypotheses. We evaluate the consequences of modeling pattern variants, once with the use of architectural primitives, generic and specialized pattern participants, and once without, respectively for the treatment and the control group. Both dependent variables are measured

Name of the variable	Class/Entity	Type of attribute(internal/external)	Scale	Measurement Unit	Range	Counting Rule
Modeling Pattern Variants	Architectural patterns	Internal	Interval	Numeric	1 to 10	Score
Architecture decomposition	Architectural patterns	Internal	Interval	Numeric	1 to 10	Score

Table 5.5: *List of dependent variables*

according to an interval scale with values ranging from 1 to 10 (1=poorest, 10=very good). The selection of the interval measurement scale was based on the measurement scales documented in [78].

5.6.3 Experiment Design

Each of the subjects participating in the experiment was specifically asked to use architectural patterns and pattern variants for designing software architecture. Two balanced groups with comparable skill levels were formed to serve the purpose. The assessment of subjects skills was performed by providing a pre-experiment questionnaire to students where they were asked to provide information about their architecture design experience, and educational background as further discussed in Section 6.4.4.

5.6.4 Subjects

The subjects in the experiment were 23 computer science graduates with experience ranging from fresh post-graduates to semi-experienced software architects. Most of the subjects had previously passed a Software Architecture course, participated in architecture design activities, and were familiar with different design diagrams like class diagrams, component diagrams, sequence diagrams etc. Some of the participants had also passed a software patterns course, where they were instructed about the use of patterns as solutions to design problems, consideration of alternate patterns, integrating patterns, and patterns influence on quality attributes. Before assigning subjects to the control and treatment groups, we determined the background knowledge and software architecture design experience of the subjects through a pre-experiment questionnaire. Among the 23 subjects participating in the experiment, there were 10 PhD candidates and 13 master students. Fig. 5.6 shows the subjects experience in architecture design and modeling patterns, as well as their understanding of patterns.

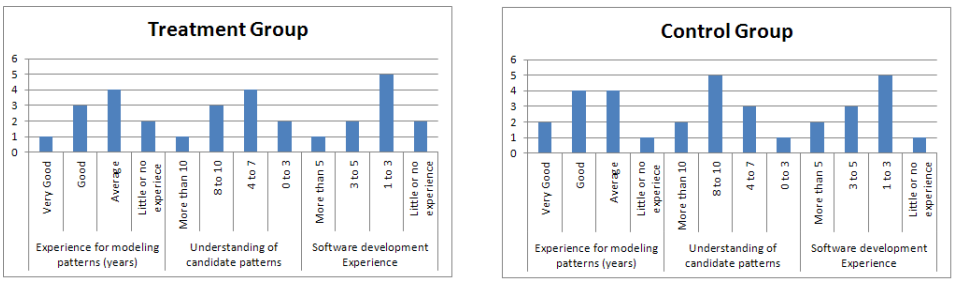

Figure 5.6: *Subjects experience and knowledge of patterns*

Experiment Material	Control Group	Treatment Group	Description
Requirements Specification	Yes	Yes	A document listing functional and non-functional requirements to be implemented in the resulting software architecture.
Candidate Patterns	Yes	Yes	A handful of architectural patterns and pattern variants
Design decisions template	Yes	Yes	A template to document important design decisions e.g. pattern selection as a solution to a design problem
Architectural patterns description	Yes	Yes	Description of selected architectural patterns in the form of design problem and associated design solution as documented in [5]
Architectural Primitives	No	Yes	Primitives description and pattern variants to primitives mapping table
A UML-based pattern modeling tool	Yes	Yes	A basic UML modeling tool
Questionnaire	Yes	Yes	Pre-experiment questionnaire to know the background of participants and post-experiment questionnaire to seek participants feedback to the experiment

Table 5.6: *Experiment Material*

5.6.5 Objects

The subjects were provided with a Software Requirement Specification (SRS) of a medium scale system, a list of candidate architectural patterns, access to a UML modeling tool, and a number of quality requirements. Additionally, handouts containing the description of several pattern variants was provided to the subjects. All architectural patterns were alphabetically indexed on a separate sheet with page numbers for easy referencing for the subjects to search and read the description of pattern variants.

Table 5.6 presents the material used in the experiment.

5.6.6 Instrumentation

Before the start of the experiment, both groups were given sufficient time to read the SRS document and ask questions to ensure their understanding of requirements. Additionally, the treatment group was provided with the architectural primitives description and patterns-to-primitives mapping document. To guide subjects for documenting the design decisions, a simple template was provided where subjects were asked to document the decision number, a short description of the design decision and the rationale for supporting the design decision.

5.6.7 Data Collection Procedures

After an introduction of the system and a question/answer session, the subjects had two hours and fifteen minutes to design the architecture. Furthermore, the subjects were requested to fill out a post-questionnaire to document their feedback about the experiment, knowledge of the listed candidate architectural patterns, and issues in using pattern variants for designing software architecture. Additionally, the treatment group was asked to document how helpful they found the proposed approach for modeling patterns. The architecture design document, design decisions document and post-questionnaires were collected from the subjects after the experiment.

5.6.8 Analysis Procedure

To analyze the data, we requested three expert reviewers to give their judgement upon the selected aspects of the software architectures. The external reviewers had several years of archi-

tecture design experience. One of the reviewers is a professor is the field of software engineering in a university in USA, the 2nd reviewer is a assistant professor in a university in China, and the 3rd reviewer is a postdoc researcher in a university in Netherlands. The information about the subjects allocation to the treatment and control groups, and treatment information was not revealed to the external reviewers. The expert reviewers were asked to provide both the grades and their feedback for each architectural design. For this purpose, a set of architecture evaluation criteria was provided to the reviewers to document their feedback. However, reviewers' identity and final results were not revealed to each other until the final data was collected.

To perform the statistical analysis of the data gathered from the expert reviewers feedback, we performed Levene's test [79] and t-test [80] to determine whether the differences in mean values calculated between the groups are significant. The Levene's test is used to check if the two groups have equal variances for the selected dependent variable. The t-test is used to measure whether the found differences are statistically significant [80]. The t-test calculates the chance that similar results will be produced when the experiment is repeated (i.e. the chance that mean values differ for control and treatment groups).

5.6.9 Validity Evaluation

We improved the reliability and validity of the experiment and data collection in two ways. First, by performing a pilot run of the experiment with one subject several days prior to conducting the experiment and taking feedback from the subject about any issues in understanding and executing the plan. This subject did not participate in the real experiment execution. Secondly, we ensured that one of the authors was available to the participants during the entire experiment, in case they faced any issues like understanding the design decisions template, availability of paper sheets etc.

5.7 Execution of the Experiment

This section discusses the instantiation of samples, randomization, instrumentation, execution of the experiment, data collection, and validation of results.

5.7.1 Sample

- *Blind experiment:* The subjects in the experiment were not told about the treatment i.e. we performed a blind experiment.
- *Blind task:* To ensure that all participants had the same knowledge of the system to be designed, the system description for which the architecture has to be designed and treatment information were kept secret until the day of exercise.
- *Technology restriction:* To make sure that the use of technology did not influence the results, all students were provided with a same basic UML-based architecture design tool for modeling pattern variants.

5.7.2 Preparation and Data Collection

The preparation and data collection went smoothly according to the experiment design described in Section 6.4 and Section 6.4.7 respectively.

5.7.3 Validity Procedure

No major problems were encountered during the execution of the experiment. One participant faced issues in using the tool in Linux environment. The subject was briefly consulted and guided appropriately.

5.7.4 Statistical Analysis of the data

With the availability of a limited number of subjects for software architecture design experiments, we believe it is important to obtain maximal information from the data gathered to draw any conclusion. The t-test, Levene's test, and intraclass coefficient tests are used to analyze the numerical data and subjective analysis is performed to interpret the results.

The t-test aims at hypothesis testing to answer questions about the mean of the data collected from two random samples of independent observations. The Levene's test is performed for equality of variances among the control and treatment groups. For the Levene's test, if the significance value is less than or equal to 0.05, then equal variances is not assumed or else the variance for both groups is considered to be equal. Separate graphs are used to present data generated from the resulting software architectures. Furthermore, to analyze the level of agreement or disagreement between external reviewers in assigning grades to software architectures, the Intraclass Correlation Coefficient test is used. The test is a general measurement of agreement between reviewers. The Coefficient represents agreements between two or more reviewers on the same set of subjects. The statistics (using demographics and tables) show the difference in the results between the control and treatment groups as shown in Appendix C and Appendix D.

5.8 Results of the Experiment

In this section, for each dependent variable, we document the number of subjects (N), the mean(M), the standard deviation (SD), and the standard error of mean (SEM) of the samples. The t-test is performed to test if the null hypotheses can be rejected.

5.8.1 Modeling Pattern Variants

Modeling pattern variants within a software architecture often requires new components, communication links, or some participants assigned additional responsibilities. The appropriate modeling of pattern variants depends on how correctly and explicitly these tasks are performed by software architects. Table 6.5 shows the mean, standard deviation and standard mean error for the control and treatment groups. There is a significant difference between the resulting mean values for the two groups (control group = 5.2 and treatment group = 6.7), which shows the better performance of the treatment group for modeling pattern variants as compared to the control group.

The Sig. value from Levene's test is greater than 0.05, as shown in Table 6.6, which shows almost equal variances among the control and treatment groups. We perform the t-test to analyze the data gathered for the 'pattern integration' aspect. Table 6.6 shows the statistics of the data. The t-test with 21 degrees of freedom(df) generates p value equal to 0.005, which is considered to be statistically significant. The p-value 0.005 shows more than 98 percent confidence that the treatment group performed better as compared to the control group.

	Group	N	M	SD	SEM
Modeling pattern variants	Control	12	5.2	1.1	0.30
	Treatment	11	6.7	1.1	0.31
Architecture decomposition	Control	10	4.8	1.0	0.31
	Treatment	11	6.5	1.1	0.33

Table 5.7: *Statistical results for the two variables (to be revised with additional data)*

5.8.2 Architecture Decomposition

An important aspect for modeling architectural patterns is decomposition of software architecture into manageable components and sub-components. Effective modeling of pattern variants can result in well partitioned software architecture [5]. Table 6.5 shows the mean, standard deviation and standard mean error of data collected for the control and treatment groups. There is a significant difference in the resulting mean value for the two groups (control group = 4.8 and treatment group = 6.5), which shows the better performance of the treatment group for decomposing the software architecture as compared to the control group.

		Levene's test for equality of variances	t-test for equality of means			
		Sig.	t	df	p	Mean Diff.
Modeling Pattern Variants	equal variance assumed	0.52	3.1	21	0.005	1.5
Architecture decomposition	equal variance assumed	0.27	3.7	21	0.001	1.7

Table 5.8: *T Test results for the two variables(to be revised with additional data)*

The Sig. value from the Levene's test is 0.27 which shows high level of homogeneity in variance between both groups [79]. We perform t-test to analyze the data gathered for the 'architecture decomposition' variable. Table 6.6 shows the statistics of the data. The p value equals 0.001, which is considered to be statistically significant [80]. The p-value 0.001 indicates the probability that 1 in 100 randomization of subjects can lead to different results [80], which is statistically negligible and we can be confident that the treatment group performed better as compared to the control group. The t-test value of 3.7 indicates that the treatment group mean is greater than the control group mean as documented in [80].

5.8.3 Data set reduction

As the subjects were specifically asked to model architectural patterns in their assignment, the exclusion criteria was based on the use of at least 4 patterns or any data point that is indicated as a outlier using the SPSS tool [80]. Figure 8.4 and Figure8.5 in appendix C show the data plot graphs for overall mean scores obtained by individual subjects. There was no outlier identified in the control and treatment groups.

5.8.4 Hypotheses Testing

Figure 6.15 shows the average score for both groups w.r.t the two aspects considered in this study. The two aspects considered in this study have p-values of 0.001 and 0.005 which are considered statistically significant to reject the null hypotheses.

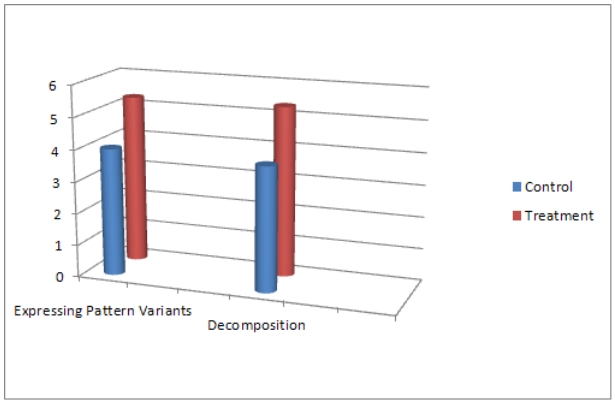

Figure 5.7: *Average scores obtained by the control and treatment groups*

5.9 Interpretation

5.9.1 Evaluation of qualitative data and implications

We performed analysis of the qualitative data received from the expert reviewers and participants, in addition to the statistical analysis performed in the previous section. The qualitative data was gathered in two forms: feedback from the participants in the post-questionnaires, and expert reviewers feedback regarding individual software architecture documents. The analysis of the qualitative data can provide additional information to assist with the interpretation of quantitative results as presented here:

- By mapping the post-experiment questions about understanding of the listed candidate architectural patterns, it was noticed that the participants in the treatment group with better understanding of the listed candidate patterns managed to produce better quality architecture as compared to the participants with similar level of expertise in the control group.
- For the control group, in a couple of cases, the reviewers were unable to identify the presence of a certain pattern variant in the software architectures. However, this was less an issue for the treatment group. The explicit representation of patterns in a software architecture help stakeholders to better understand the architecture and reason about the quality attributes deemed in the resulting software architecture [5]. The use of a pattern modeling approach, such as the once presented in this study, can help inexperienced software architects to better express pattern variants within software architecture.

5.9.2 Limitations of the Study

Threats to internal validity:

Internal validity is the degree to which the values of dependent variables can be attributed to the experiment variables.

- *Balanced groups:* In order to avoid bias in allocating participants to the treatment group, we assigned participants to each group randomly based on their expertise level. For instance, if there were 6 subjects with same level of expertise(i.e. experience, education background etc.), 3 were randomly picked for the treatment group and the other 3 for the control group.

- *Use of statistical method:* Another threat to validity is the selection of appropriate statistical method for data evaluation. We addressed the issue by sending data to an expert in the field of statistics and by studying alternate statistical methods to pick one that best fits the nature of data gathered after the experiment i.e. interval scale, scores ranging from 1 to 10 etc.

- *External reviewers bias:* There is a possibility that external reviewers may be biased in grading one or more architectural aspects considered in this study. This is because the expert reviewers may interpret a selected architectural aspect differently e.g. the design comprehensibility aspect may be interpreted differently by different reviewers. In an effort to reduce the impact of reviewers bias on final results, the selected aspects were discussed with the reviewers. A brief description of the aspects was then provided to the three reviewers.

Threats to external validity: External validity concerns how far the results of a study can be generalized.

- *Generalization:* The subjects who participated in the experiment (post-graduate students) are unlikely to be representative of experienced industrial software architects. However, Sjoberg et al. in [81], have also suggested that graduate students of computer science be considered as semiprofessionals and hence are not so far from practitioners. The experiment results encourage us to further exploit the use of primitives, generic and specialized participants for modeling pattern variants in industrial experiments.

- *Time constraint* Software architecture design is a lengthy and complex activity and not all of the architectural aspects (i.e. architectural views, detail component partitioning etc.) can be addressed in a limited time frame. However, subjects were asked to perform a limited task, to model pattern variants for designing software architectures. In the design of the experiment, we considered two hours and fifteen minutes sufficient for subjects to come up with reasonable architecture. The decision to allocate 2 hrs 15 mins time slot for each group was verified by a pre-experiment pilot run of the study.

- There is a risk that the three expert reviewers may significantly differ in rewarding grades to a specific software architecture. To avert this risk, we performed the Intraclass Correlation Coefficient test to identify major differences in grades. The test was used to identify the degree of homogeneity in grades. The tables in Appendix D provide the results of performing the Intraclass Correlation Coefficient test, which shows acceptable level of difference in homogeneity of grades.

5.10 Conclusions

The architectural pattern variant modeling approach that uses the primitives and pattern-specific architectural elements in combination offers a systematic way to model pattern variants. With examples and results from a controlled experiment, we illustrated the approach for modeling architectural patterns variants. The approach to use generic and specialized pattern participants in conjunction with architectural primitives offers reusability support by providing a vocabulary of design elements that entail the properties of known pattern participants that are extensible enough to be specialized for modeling several variants of a pattern.

Parts of this chapter has been published as:

1) A. W. Kamal and P. Avgeriou – *"Mining Relationships Between the Participants of Architectural Patterns,"* proceedings of the 4th European Conference on Software Architecture (ECSA), pp. 401 - 408, Springer-Verlag, August 23, 2010, Copenhagen, Denmark, Lecture Notes in Computer Science, Springer.

2) A. W. Kamal, P. Avgeriou, and U. Zdun – *"The Use of Pattern Participants Relationships for Integrating Patterns: A Controlled Experiment"*, Published in the Journal of Software Practice and Experience, Wiley InterScience, 2011.

Chapter 6

The Use of Pattern Participants Relationships for Integrating Patterns: A Controlled Experiment

Abstract

Architectural patterns are often applied in combination with related patterns within software architectures. The relationships among architectural patterns must be considered when applying a combination of patterns into a system; for example the way the Model-View-Controller uses the Observer pattern to implement the change propagation mechanism needs to be carefully designed. However, effective integration of architectural patterns within software architectures remains a challenging task. This is because the integration of any two architectural patterns can take several forms. Furthermore, existing pattern languages define generic and abstract relationships between architectural patterns without going into detail about associations among the participants of architectural patterns. In this chapter, we propose to address the pattern integration issue by discovering and defining a set of pattern participants relationships that serve the purpose of effectively integrating architectural patterns. Our findings are validated through a controlled experiment, which provides significant evidence that the proposed relationships support inexperienced designers in integrating patterns.

6.1 Introduction

Over the past decade, architectural patterns have increasingly become an integral part of software architecture design practices [5]. Architectural patterns often specify solutions to recurring design problems by describing essential components, their responsibilities and relationships [59]. In practice architectural patterns are seldom applied in isolation to a software architecture, as a single pattern may not suffice to fully resolve a design problem at hand. For instance, the Client-Server and Broker patterns are often used in combination to design distributed systems [5]. It is commonly accepted that patterns are somehow connected to each other giving them the potential to solve larger design problems [8] [59].

There have been several approaches in the patterns community that aggregate a number of patterns that define to some extent relations between those patterns. We characterize these approaches into four major categories:

- *Architectural pattern languages* define a network of patterns, connected through specific relationships [41], forming a graph of nodes where each node represents a pattern [59]. Pattern languages are the most common and well-known form, used by the software patterns community for defining relationships among architectural patterns. Several pattern languages have been documented in the literature e.g. pattern languages for distributed computing [59], domain specific pattern languages [5], architectural views-specific pattern languages [41] etc.

- *Pattern catalogs*, which may be organized according to different categories like patterns for object oriented frameworks [82], patterns for enterprise computing [83], patterns for security [6], patterns for user interface design [84]. Patterns catalogs often list patterns alphabetically or according to some categorization, but they do not always describe the relationships between patterns. Patterns in a pattern catalog do not form a pattern language because their contexts do not weave them together [59].

- *Pattern compounds* capture recurring use of a set of patterns that are often used as a single decision to solve a recurring design problem [59]. For instance, the Batch Method [59] and Iterator [59] are often used together leading to the commonly used term BatchIterator.

- *Pattern sequences* document the possible successive application of patterns for designing software architectures [85]. For instance, first the Iterator pattern can be applied to provide the notion of a traversal position, then, the Batch Method is applied to define the style of access on a component [8].

However, these approaches do not support the effective integration of architectural patterns within software architecture, for two main reasons:

- Existing pattern languages, pattern compounds and pattern sequences document associations between patterns at a generic level but do not go into details concerning the relationships between the pattern participants[1]. For instance, a pattern language may suggest that the communication between Client and Server [5] can be mediated through a Broker [5] and hidden by a Proxy [59]. But it does not elaborate on how the participants of these three patterns will collaborate in order to achieve the envisioned goal. The relationships among architectural pattern participants are important to effectively address the extended set of requirements that mandate the combination of two or more patterns. The details of such relationships between participants concern for example how participants of related patterns overlap, interact, or override other participants in the resulting software architecture.

- The integration of two selected architectural patterns does not always result in one particular solution but leads to several possible design solutions depending on the system context at hand. In other words, pattern-to-pattern relationships are not always fixed but may entail a great deal of variability. For instance, to model interactive applications, the Model-View-Controller (MVC) [5] and Layers [5] pattern can be combined in several different forms. In the 2-tier layered variant, the presentation layer consists of the View and Controller participants while the application logic layer owns the Model participant. However, in the 3-tier application architecture, the View may correspond to user interface layer, Controller correspond to business layer, and Model correspond to data logic layer. This variability in pattern combinations is currently not explicitly addressed by existing pattern languages.

In this chapter, we aim at supporting architects and designers in integrating patterns by using relationships at the level of pattern participants. The relationships were discovered after reviewing architectural patterns modeled in several industrial software architectures and pattern integration examples documented in the literature. The notion of pattern participants relationships was first proposed by us in [33]. Our current work provides detailed documentation

[1] By the term pattern participants we refer to the architectural elements within the solution of architectural patterns e.g. pipes and filters are pattern participants of the solution specified by Pipes and Filters pattern.

of discovered relationships and is supported by evidence in a controlled experiment. The documentation of pattern relationships at the pattern participants level contains several possible associations between architectural patterns, which correspond to alternative design solutions. Furthermore we have validated our approach through a controlled experiment where we investigated the effectiveness of using pattern participants relationships in integrating architectural patterns. We advocate that the use of pattern participants relationships a) leads to appropriate integration of architectural patterns, b) improves design comprehensibility, c) helps architects to better document design decisions, and d) assists in decomposing software architectures.

The remainder of this chapter is structured as follows: In Section 6.2, we describe related work in the field of architectural patterns relationships. Section 6.3 describes our effort for identifying relationships among architectural patterns participants and lists a set of pattern participants relationships. Section 6.4 provides the description of the controlled experiment that was conducted to test the effectiveness of using pattern participants relationships for integrating architectural patterns and Section 6.5 documents the execution of the experiment. Section 6.6 presents statistical results from the controlled experiment. Section 6.7 interprets qualitative data gathered after the experiment and discusses the possible threats to the validity of the results. The study is concluded in Section 6.8.

6.2 Related Work

In this section we discuss some of the work done by other researchers in the area of relating architectural patterns.

Zimmer [86] classifies the relationships between several design patterns. He categorizes pattern relationships into three categories namely: *uses* where a pattern A must use a pattern B, *similar* where a pattern A is similar to pattern B, and *combined* where two patterns can be applied as one design solution to a design problem. However, the classification of relationships addresses only design patterns where each pattern is represented as a single unit or object. His work addresses the abstract relationships between patterns and the pattern links in the context of a pattern language. Our work is aimed at documenting relationships between the participants of architectural patterns and the way patterns are combined in real software architectures. Thus, we provide a more fine grained approach to associate patterns using pattern participants relationships for effectively integrating architectural patterns.

Fayad et. al. [87] have proposed the concept of *stable software patterns*. The process of developing stable software patterns involve four main steps: *developing stable patterns*, *documenting stable patterns*, t*esting/validating stable patterns*, and *applying stable patterns*. For each of the four steps there are different sets of patterns that interact together to accomplish the goal of the step. However, their work is more focused on software stability concepts [88]. Our contribution differs from this work as our focus lies in discovering relationships between patterns at a rather detailed level of abstraction i.e. between the participants of patterns in real software architectures which is not addressed before. In relation to their work, if more relationships are identified, our work can be used in the *applying stable patterns* step of their approach.

Buschmann et al. [89] document three kinds of pattern relationships: pattern complements, where one pattern competes with another by providing an alternative solution to a specific problem; pattern compounds that relate two or more patterns for their use as a single decision to solve a design problem; and pattern sequences that describe the progression of patterns by having predecessor patterns forming part of the context of successive patterns. However, these types of pattern relations are defined at an abstract level and do not provide concrete relationships between the participants of related patterns. Our work aims to fill this void by

Approach	Granularity	Application	Scope	Contribution
Relationships categorization [86]	Classes, objects	Design patterns	Object-Oriented system design	Abstract relationships between design patterns
Software stability concerns [87]	Design activity nodes	Processes	Software stability concepts	Stable software development
Pattern relationships types [89]	Architectural elements	Architectural and design patterns	General	Generic pattern language
Architectural concerns [84] [6]	Components and connectors	Architectural patterns	Quality-attributes driven design	Quality-attributes specific pattern languages
Grouping patterns based on problem-domain [59]	Components and connectors	Architectural patterns	Distributed systems architecture design	Domain-specific pattern language
Formal approaches to modeling patterns [90] [91]	Classes, nodes, objects, function calls	Design patterns	General	Ontology-based pattern modeling
Empirical research [92] [93]		Architectural patterns, Design patterns	General	Statistical results for applying patterns
Pattern participants relationships	Components as participants of architectural patterns	Architectural Patterns	General	Relationships between pattern participants and statistical results for integrating patterns

Table 6.1: *Overview and comparison of related work*

addressing pattern relationships at a detailed level of granularity i.e. at the level of pattern participants.

Some work has been done on proposing pattern languages that address specific architectural concerns such as pattern languages for usability [84], pattern languages for concurrency [6], pattern languages for performance-critical systems [59] etc. However, these languages provide relationships that address specific architectural concerns they relate to and do not address the relationships among participants of related architectural patterns. For instance, the 'event handling' relationship [59] between the Reactor and Leader/Followers patterns does not specify the participants of the two patterns that need to be combined for designing an event handling solution.

Buschmann et al. [59] present a pattern language for distributed computing that includes 114 patterns grouped into 13 problem areas. The problem areas address technical topics related to building distributed applications e.g. Event Demultiplexing, Concurrency, Synchronization etc. This pattern language serves as an overview of the selection and use of related architectural patterns to solve design problems in specific problem areas. However, the language in itself presents architectural patterns as components, objects and entities linked through generic textual expressions. For instance the Model-View-Controller has a 'request handling' relationship with the Command, Command Processor, Application Controller, and Chain of Responsibility patterns. Similar to the previous cases, relationships between participants are not defined.

In our previous work [41], we have documented relationships among architectural patterns in different architectural views that show specific aspects of systems like data flow view, interaction decoupling view etc. We had focused on providing rich pattern-to-pattern relationships (e.g. communication between Layers may use Pipes and Filters), and not on relationships among participants of architectural patterns.

There have been several attempts at introducing formal representations in the design patterns area such as pattern representation supported with ontologies [90] or formally specifying design patterns solution (see for instance [43]). The ontologies concept is used primarily to describe the structure of source code, which is done according to a particular design pattern. The

work in the field of ontology-based pattern modeling mostly covers [91]: a) creation of standardized templates for the description of ontology-based design patterns, b) creation of functional and generalized methods for users with different level of expertise in pattern reuse, and c) creation of techniques and tools for supporting a semi-automatic or automatic pattern selection. However, most of the ontology based pattern modeling approaches work at detail-level design issues such as functional calls, parameter passing etc. Similarly, the approaches to formally specify design patterns have not gained much momentum in recent years mainly because of their complexity and their resulting limitations regarding their practical use. Moreover, these approaches have not been used for architectural patterns or whole pattern languages, like our relationships, but just for some isolated patterns. Our work focuses on relationships between the participants of several architectural patterns at the architecture design level offering different possibilities to combine architectural patterns which is not yet fully addressed by existing ontology-based pattern modeling approaches and formal pattern specification approaches.

There have also been several attempts for specifying existing Architecture Description Languages (ADLs) [16] or proposing new ADLs such as ADLs proposed as extensions of UML [46], usually in the form of profiles. Most of these ADLs treat architectural patterns as first-class entities and provide tool support for modeling patterns. For instance, the ACME [16] provides built-in templates that can be used to model patterns. However, extensive design effort is required to merge, remove, override the participants of related patterns for integration. Our approach aims at more flexibility by providing a wider range of lower-level relationships that once supported by an existing ADL, can be used for effectively integrating architectural patterns.

Some researchers have performed empirical studies for the use of architectural patterns in designing software architecture [92], as a mechanism to capture design decisions [93], and as solutions to satisfy specific quality attributes [63]. However, to the best of our knowledge, no work has been done so far to evaluate the effectiveness of using pattern languages for integrating architectural patterns within software architectures.

Table 6.1 gives an overview of the related work, and how it compares to the approach presented in this chapter.

6.3 Mining Pattern-Participants Relationships for Modeling Patterns

The relationships presented in this section are based on the study of software architectures from 32 industrial software systems [94], pattern integration examples documented in the literature [5][59][8], and patterns presented in workshops and conferences [12]. The patterns integrated within real software architectures are analyzed to discover the relationships between the participants of related architectural patterns. In the following sub-sections, we describe the approach for mining pattern participants relationships, a template to document the discovered relationships, and finally present all relationships discovered during this study.

6.3.1 The mining process

The underlying idea behind our approach is that various architectural patterns can be effectively integrated using a set of 'pattern participants relationships'. The relationships serve as a basis for identifying the participants of architectural patterns that share, overlap, or override other participants of related architectural patterns. Specifically, we followed three steps to mine pattern participants relationships:

- We started by identifying architectural patterns from architecture design diagrams by following the pattern mining process defined in [63]. Subsequently we identified the participants of the discovered patterns that associate with participants of other patterns within the software architectures. We documented such associations.
- We used the software architecture design documents of several industrial software systems to read the description of architectural patterns integrated for designing such systems. The relationships between the participants of such patterns were identified and documented.
- We studied the pattern integration examples documented in the literature to look for pattern participants relationships. We came across several pattern integration examples in [5], [59], [8], etc.

6.3.2 Template for pattern participants relationships documentation

Each pattern participant relationship discovered during this study is documented according to the following template:

- *Name*: Short intent of the relationship and its name.
- *Issue*: Brief description of a design issue for integrating patterns.
- *Definition*: Description of the pattern participant relationship in the context of a combination of architectural patterns.
- *Known uses in patterns integration*: Three known uses of the relationship in combinations of architectural patterns.
- *Example*: A pattern integration example to describe the discovered relationship.

6.3.3 Pattern Participants Relationships

The solution specified by architectural patterns is comprised of components and connectors called pattern participants [5]. However, certain patterns can be integrated in a single component or connector of another pattern and hence can be considered as participants of such patterns as documented in [63]. For instance, the Asynchronous [59] pattern can be combined with the Pipe participant of the Pipes and Filters pattern for defining asynchronous connections between adjacent Filters. To avoid complexity, we call such participants as patterns in the examples documented in this section. Following, we use the mining process described above to document relationships between participants of different architectural patterns.

Redundant pattern participants: absorbParticipant

Definition: An absorbParticipant relationship defines how the participants of different patterns performing similar responsibilities are integrated in a single element. In an absorbParticipant relationship, certain participants of a pattern are absorbed by the participants of another pattern to avoid redundancy.

Issue: Consider the case where two patterns consist of two or more functionally-equivalent, yet independent pattern participants. One or more such participants in a pattern, however, may become redundant and cannot be included in the software architecture as they are. The

problem, now, is the way in which the pattern integration process should deal with these redundant participants in an effective way.

Known uses in pattern integration:

- The event handling solution is present in both the Proactor [59] and Leader Follower [59] patterns.
- Both the Reactor and Proactor [5] patterns introduce their own handles for demultiplexing and dispatching events to corresponding event handlers.
- The Dispatcher participant is present in both the Acceptor-Connector [8] and Interceptor [8] patterns.

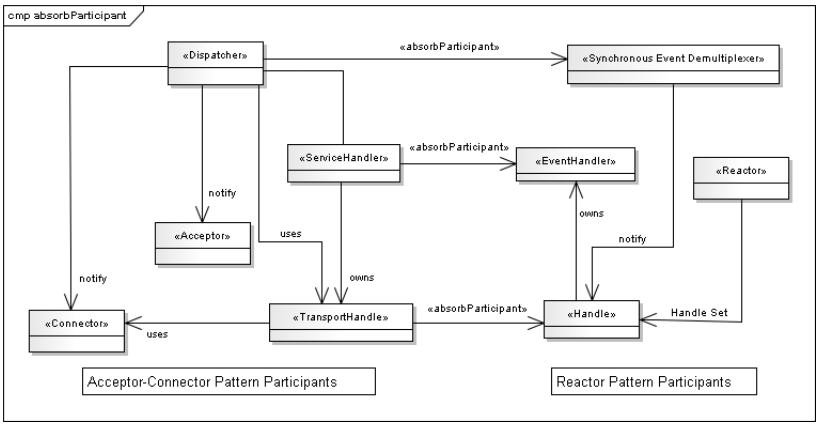

Figure 6.1: *The absorbParticipant relationship between the Reactor and Acceptor-Connector Patterns*

Example: Both the Reactor and Acceptor-Connector patterns introduce their own event handling participants for using different services [8]. The separate event handling solutions in both patterns carry redundancy in applying these patterns in combination to a software architecture e.g. the handler participant is present in both the Reactor and Acceptor-Connector patterns. Figure6.1 shows the absorbParticipant relationship between the Reactor and Acceptor-Connector patterns.

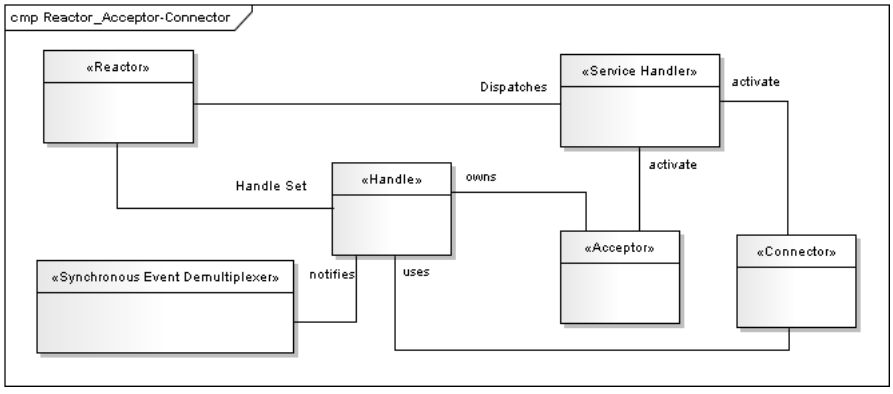

Figure 6.2: *Integrating the Reactor and Acceptor-Connector Patterns*

In the Reactor's architectural structure, for each service an application offers, a separate event handler is introduced that processes different types of events from certain event sources.

However, the Acceptor-Connector pattern can be considered as an option to implement the Reactor's event handlers. This ensures that the Reactor pattern specifies 'right' types of event handlers associated with the Acceptor-Connector pattern. In order to integrate both patterns, the overlapping pattern participants either need to be merged or participants of one pattern be replaced by the other. However, removing a specific participant within a pattern may impact the solution specified by that pattern and may require new associations between the participants of both patterns, which is not a trivial task and requires extensive design effort. Figure 6.2 shows the resulting architecture after integrating the Reactor and Acceptor-Connector patterns using the absorbParticipant relationship.

Overlapping pattern participants: mergeParticipant

Definition: The mergeParticipant relationship is used to combine one or more semantically different pattern participants into a single participant within the target pattern. Such an integration retains the structural and semantic properties of individual participants into the target element. The mergeParticipant relationship is different from the absorbParticipant relationship where participants performing similar responsibilities are absorbed (i.e. redundant participants are virtually removed in the resulting software architecture).

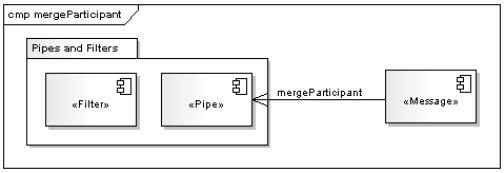

Figure 6.3: *The mergeParticipant relationship*

Issue: While integrating architectural patterns, the overlapping pattern participants problem occurs when participants present in different patterns are intended to represent the same concept and hence need to be merged in a single component. Nevertheless, the resulting component is deemed to represent the result of the merge, in the same way that functions of both participants are present in the resulting participant and not merely the increment added by a participant.

Known uses in pattern integration:

- For logic-intensive interactive applications, the Model participant of the MVC [5] pattern can merge the responsibilities of the Strategy [59] pattern.
- In the Document-View pattern variant[5], the View participant combines the responsibilities of both the View and Controller from MVC using the mergeParticipant relationship while the Document participant corresponds to the Model in MVC.
- The Master participant within the Master-Slave [59] pattern can merge the Strategy [59] pattern for configuring the varying strategies without affecting the slaves.

Example: In a distributed data processing arrangement, pipes are realized as a form of messaging infrastructure to pass data streams between remote filters. Such a design supports flexible redeployment of filters in a distributed pipes and filters architecture. In such a structure, the message pattern can be merged with the Pipe participant to setup messaging pipes between filters. Figure 6.3 shows the mergeParticipant relationship between the Pipe participant and the Message pattern while Figure 6.4 shows an example architecture after integrating the Pipes and Filters pattern with the Message pattern.

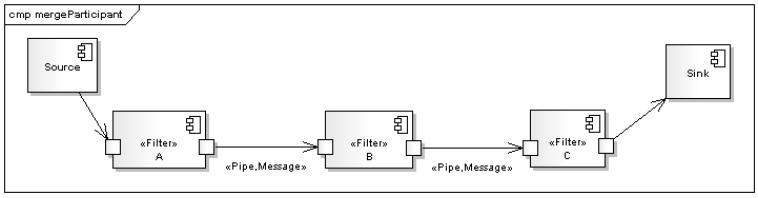

Figure 6.4: *The mergeParticipant relationship Example*

Modeling patterns within the participant of a target pattern: importPattern

Definition: importPattern is a relationship where the participant(s) of a target pattern import all participants from a source pattern. This means all participants of a pattern are modeled within the participant of another pattern. The importPattern relationship is similar to Package import in UML [64], Family import in ACME [62], etc.

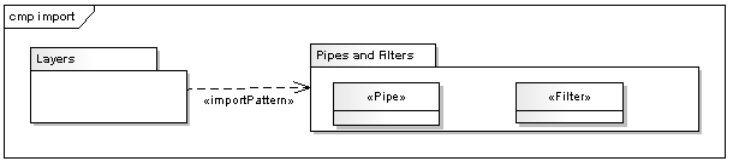

Figure 6.5: *The importPattern relationship*

Issue: In certain cases of pattern integration, it is possible that one pattern acts as a solution participant of another pattern to solve a design problem at hand. However, one challenge to model such a solution is that the imported pattern must not overwrite the target pattern as both work as complementary solutions to a design a problem and not as alternatives.

Known uses in pattern integration:

- The Dispatcher participant within the Client-Server [5] pattern imports the Activator pattern to activate/de-activate services running on different servers.
- In a distributed software architecture design, the Broker [5] hides and mediates all communication between the objects or components of a system. A Broker can import the Requester pattern for its internal implementation in order to forward requests from a client to associated components.
- The Server participant within the Client-Server pattern can itself be internally partitioned into several layers by importing the Layers pattern.

Example: Individual layers in the Layers pattern can import other patterns for their complete implementation. For instance, a single layer may be implemented as a data processing layer using the Pipes and Filters pattern as shown in Figure 6.5 while an example of the import-Pattern relationship is shown in Fig 6.6.

Modeling participants within another pattern participant: importParticipant

Definition: An importParticipant is a relationship where participants of the target pattern import *specific* participants from the source pattern.

Issue: Similar to the problem addressed in the importPattern relationship, the import of specific participants into target pattern must not replace the target pattern's participants. For

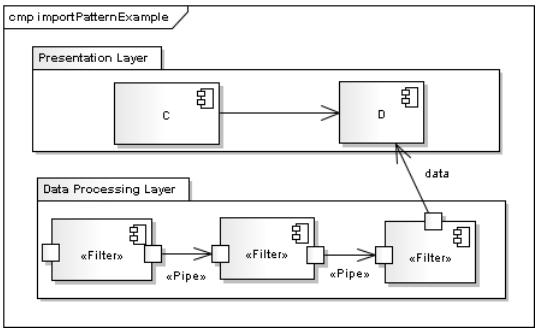

Figure 6.6: *The importPattern relationship Example*

instance, integrating two or more objects or classes, defined as pattern participants, must not override each other in the resulting architecture.

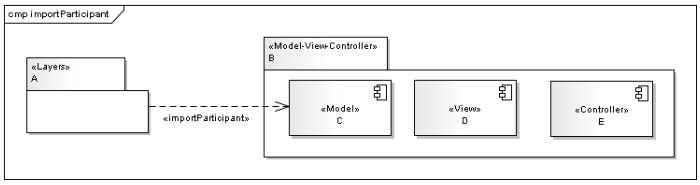

Figure 6.7: *The importParticipant relationship*

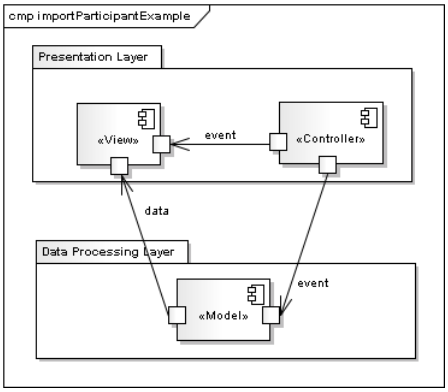

Figure 6.8: *The importParticipant relationship Example*

Known uses in pattern integration:

- A specific Layer may import the Model participant of the MVC [5] pattern.
- The Broker [5] and Reactor [8] patterns can be integrated to design event-driven software architecture. In this particular example, the Request Handler participant of the Broker pattern imports the Event Handler participant of the Reactor pattern to handle multiple event sources simultaneously.
- When modeling the Half-Sync/Half-ASync [59] and Reactor [8] patterns in combination, the Quering Layer participant of the Half-Sync/Half-ASync pattern imports the EventHandler participant of the Reactor pattern to synchronize the invocation of services.

An example to describe the issue: Individual layers in the Layers pattern can import other patterns's participants. The Presentation layer can import only the View and Controller participants of the MVC pattern while the Model participant of the MVC pattern resides in the data logic layer as shown in Figure 6.7 and Figure 6.8.

Participants make use of related pattern participants: employ

Definition: Employ is a relationship where participants of a pattern generally make use of another pattern for their complete implementation. Patterns using the 'employ' relationship are often applied together within software architectures where one pattern 'makes use of' another pattern to fulfill specific design needs. Frequent use of these patterns together helps associate the related participants of such patterns.

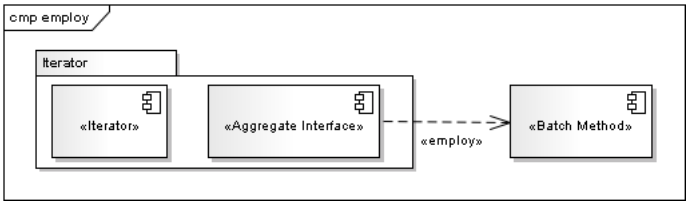

Figure 6.9: *The employ relationship*

Issue: The loose dependency relationship described above is not explicit in current pattern relationship approaches making it difficult for software architects to combine related architectural patterns.

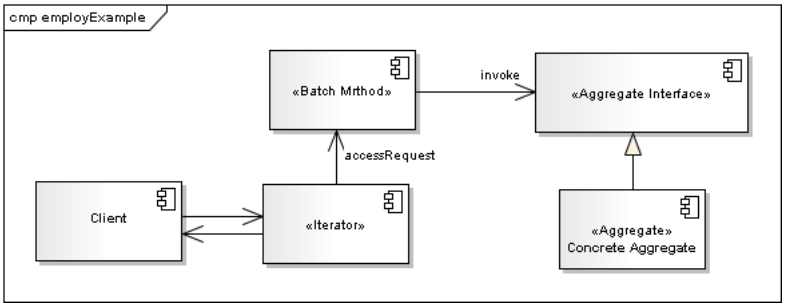

Figure 6.10: *Integrating the Iterator and Batch Method Patterns*

Known uses in pattern integration:

- The MVC [5] pattern employs the Observer [5] pattern to implement the change propagation mechanism.
- The Command [59] pattern implementation using the Composite [59] pattern is so common that it is often considered as single design solution to which the name Composite-Command is also used.
- The Broker [5] pattern often employs the Receiver and Invoker patterns so that clients can receive data and invoke services effectively.

Example: The Iterator pattern often employs the Batch Method pattern that supports access to aggregate elements without causing performance penalties and unnecessary network loads

when the Iterator is remote to the aggregate [59]. The combined use of both patterns often leads to the term 'Batch-Iterator' pattern in the literature [59]. However, both patterns can be applied independently to a software architecture to solve specific design problems. Figure 6.9 shows the employ relationship among the participants of the Iterator and Batch Method patterns. The resulting architecture after integrating the Iterator and Batch Method patterns is shown in Figure 6.10

Strong coupling between pattern participants: depends

Definition: The depends relationship shows the need of pattern participant(s) to use another pattern for their complete implementation. In contrast to the employ relationship, the depends relationship is a strong dependency of a pattern's participants on another pattern where, in practice, the participants of the source pattern always use the participants of the target pattern.

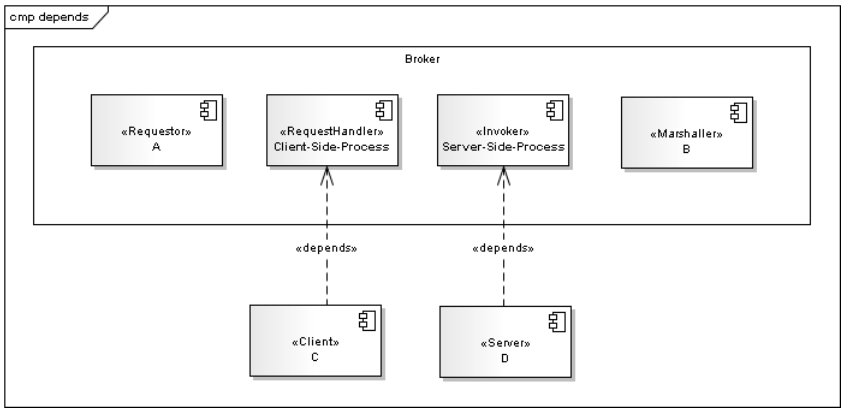

Figure 6.11: *The depends relationship*

Issue: The strong coupling between the participants of relating patterns require that the participants from the source and target patterns must be present in the resulting software architecture. The strong dependency relationship between the participants of related patterns is not explicit in current pattern languages.

Known uses in pattern integration:

- The Control participant within the PAC [5] pattern depends on the use of the Mediator [59] pattern for co-ordinating with other PAC agents.
- The Microkernel [5] pattern is often modeled as three layered architecture i.e. the Microkernel depends on the Layers pattern.
- The Reflection [5] pattern is modeled as a two layered architecture using the Layers [5] pattern: a meta level contains the metaobjets, a base level the application logic.

Example: The Broker pattern separates and encapsulates the details of communication infrastructure in distributed systems. In such a structure, clients invoke remote services using the Broker as if they were local and in return receive response from servers that offer these services. In such a system context, the Broker pattern is always modeled in combination with the Client-Server pattern, as analyzed in this study, which is documented using the depends relationship between the Broker and Client-Serve patterns. The depends relationship is shown in a particular example of Client-Server and Broker patterns integration in Figures 6.11 and 6.12.

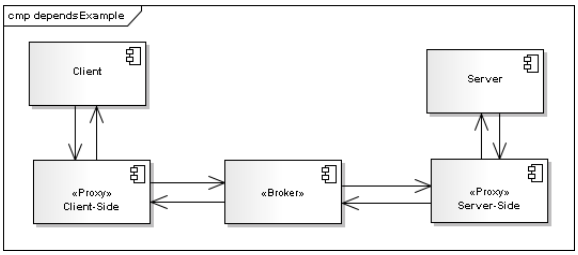

Figure 6.12: *Integrating the Broker and Proxy Patterns*

Mediator pattern participants: interact

Definition: An interact is a relationship where certain participants of the source pattern interact with the participants of the target pattern to solve a design problem. In an interact relationship, the target pattern often acts as a mediator/redirector by mediating the requests between the source pattern and surrounding architectural elements.

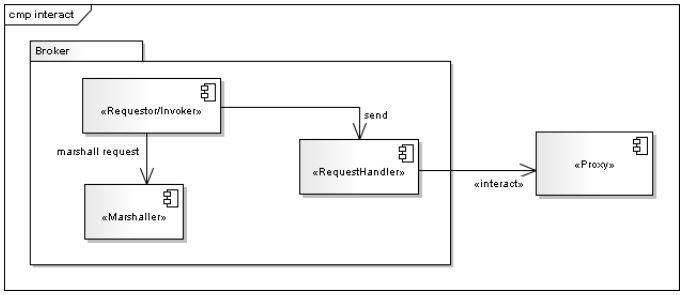

Figure 6.13: *The interact relationship*

Issue: The mediation/redirection role of the participant of a pattern to integrate related patterns requires that the communication to target components must pass through only the mediating participant. Such a relationship is challenging to identify, at pattern participants level, using current pattern relationships approaches.

Known uses in pattern integration:

- The Client-Server [5] pattern interacts with the Proxy [5] pattern for sending/receiving messages. The proxy acts as a communication redirector between Clients and Server.
- The Layers [5] pattern can use the Adapter [59] pattern that connects provided interface of the components in one layer into the interface that the clients expect in another layer, and vice versa.
- A Virtual Machine [41] interacts with the Layers [5] pattern by redirecting invocations from a byte-code layer into an implementation layer for the commands of the byte-code.

Example: The Broker [59] pattern interacts with the Proxy [59] pattern to send/receive information to/from Client and Servers. The Proxy pattern acts as a service redirection to forward requests to appropriate client/server. Figure 6.13 shows the interact relationship between the Proxy and Broker participants and Figure 6.14 shows the resulting architecture.

The intention is to use the pool of all available pattern participants relationships to integrate several architectural patterns. However, the relationships between patterns, as listed in Table 6.2, are not fixed: rather the solutions entailed by two selected patterns can be combined in

Patterns	Acceptor-Connector	Activator	Adapter	Blackboard	Broker	Client-Server	Composite	Event-Handler	HalfSync/AHalfSync	Invoker	Mediator	Layers	Microkernel	MVC	Observer	Pipes and Filters	Plug-in	Proxy	Publish-Subscribe	Receiver	Requester	RPC	Scheduler	Shared Repository	Strategy	Leader-Follower	Mediator	Proactor	Virtual Machine
Active Repository [59]															importPattern														
Blackboard [59]																	importPattern					interact							
Broker [59]						depends				employ								interact		employ	importPattern								
Client-Server [5]	absorbParticipant	importPattern		mergeParticipant	interact				absorbParticipant						importPattern			interact		importPattern									
Layers [5]			interact		importPattern,interact	importParticipant	importParticipant					importParticipant	importParticipant	importParticipant	importPattern	importPattern	importPattern	importParticipant											interact
Microkernel [59]												depends																	
MVC [59]															employ	interact							interact		mergeParticipant				
PAC [59]												mergeParticipant											interact		importPattern		depends		
Peer-to-Peer [59]						mergeParticipant										importPattern													
Pipes and Filters [5]					interact																								
Plug-in [4]											interact																		
Lin1 12 Publish-Subscribe [59]					interact											importPattern													
Lin1 34 Reactor [59]	absorbParticipant				importParticipant			importParticipant	importParticipant							importPattern												absorbParticipant	
Shared Repository [4]													importPattern			importPattern													
Proactor [59]																										absorbParticipant			
Interceptor [59]	absorbParticipant																												

Table 6.2: *Pattern Participants Relationships Discovered in Software Architectures*

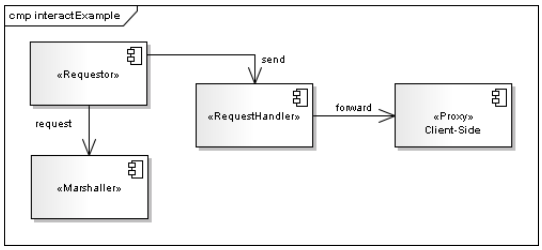

Figure 6.14: *The use of interact relationship for integrating patterns*

infinite different ways and so is the selection of relationships for integrating patterns. Thus, the decision to apply a specific relationship for integrating patterns lies with the architect who picks relationships that best meet the design needs at hand i.e. the *absorbParticipant* relationship is used only if it is required to avoid redundant participants in the resulting architecture. Using our relationships allows an architect to integrate several architectural patterns with certain level of design support w.r.t design requirements at hand.

Table 6.2 lists the mined relationships in a tabular form. We note that the set of relationships were elicited from real architectures, so they are actually practiced by software architects. However they are not explicitly documented and architects cannot reuse them but need to discover them on a case-by-case basis. By documenting them, we strive for reusability of the relationships, especially among inexperienced architects and designers. Moreover, we have previously proposed a approach, called Pattern-Driven Architectural Partitioning (PDAP)[63] that documents how a pattern may influence the use of other patterns, a pattern may specialize the use of another, or how two patterns may be alternatives, etc. A architecture design process, such as PDAP, can be successfully followed alongside the use of pattern participants relationships as the proposed relationships are aimed at complementing the existing software architecture design methods without affecting the core design activities of these methods, e.g., analysis, design, evaluation, etc. For instance, in the PDAP method, the step to partition the system by applying a combination of the candidate patterns can benefit by the use of pattern participants relationships. In the following sections, we present results from a controlled experiment where subjects had the freedom to follow any architecture design method for modeling patterns. In addition, a group of participants were provided the list of relationships to test the effectiveness of using the relationships for integrating architectural patterns.

6.4 Experiment Design

We claim that the pattern participants relationships, discussed in the previous section, help architects to effectively integrate architectural patterns within software architectures. To test this claim, we performed a controlled experiment. In this experiment, two groups of graduate students were provided with the same information for designing software architectures based on a set of requirements. In addition, one group was provided with information about pattern participants relationships. The experiment was designed according to the guidelines for conducting empirical research in software engineering [21] and the results were documented according to the reporting guidelines for controlled experiments in software engineering [27]. Moreover, the suggestions from the working group for conducting controlled experiments [95] were taken into consideration while designing the experiment.

In the following sub-sections, the design of the experiment, our hypotheses, involved subjects, variables, and the analysis of the data collected from the experiment are presented. With

the final data gathered from the outcome of the experiment, we evaluate how the use of pattern participants relationships can help architects to effectively integrate architectural patterns within software architectures.

6.4.1 Research Question and Hypotheses

To analyze the use of pattern participants relationships for integrating architectural patterns, we present the research question and construct null hypotheses (H0i) and alternate hypotheses (H1i) as explained below.

Research Question: *Does the use of pattern participants relationships help to effectively integrate architectural patterns within software architecture?*

The effective integration of architectural patterns covers several aspects of software architecture design [77]. We have selected four such aspects that are evaluated according to the following hypotheses:

Null Hypotheses:

1. *H00:* The use of pattern participants relationships does not help software architects to more accurately[2] integrate architectural patterns within software architectures as compared to integrating architectural patterns without using such relationships.

2. *H01:* The integration of architectural patterns using pattern participants relationships does not result in a more comprehensible software architecture as compared to integrating patterns without the use of such relationships.

3. *H02:* The use of pattern participants relationships does not help software architects to better document architectural design decisions as compared to documenting design decisions without using such relationships.

4. *H03:* The use of pattern participants relationships does not help software architects to more effectively partition a software architecture into components and sub-components, and assign responsibilities as compared to partitioning a software architecture without the use of such relationships.

Alternate Hypotheses:

1. *H10:* The use of pattern participants relationships helps software architects to more accurately integrate architectural patterns within software architecture as compared to integrating architectural patterns using such relationships.

   How accurately the selected architectural patterns are integrated within software architecture to solve the design problem at hand e.g. identifying the presence of redundant pattern participants, inappropriate linking of pattern participants etc.

2. *H11:* The integration of architectural patterns using pattern participants relationships results in a more comprehensible software architecture as compared to integrating patterns without the use of such relationships.

   Software architecture design comprehensibility refers to the understandability and completeness of resulting software architectures.

[2] The term accurately refers to identifying the presence of redundant pattern participants and inappropriate linking of pattern participants within a software architecture.

Name of the variable	Class/Entity	Type of attribute (internal/external)	Scale	Measurement Unit	Range	Counting Rule
architecture design experience	Software Architecture	external	ordinal	year	above 5, 3 to 5, 1 to 3, no experience	pre-experiment feedback
Experience of integrating patterns	Architectural Patterns	external	ordinal	expertise	Very Good, Good, Average, Little or no experience	pre-experiment feedback
Belief in using patterns	Architectural Patterns	external	ordinal	agreement	Very helpful, helpful, just OK, not very helpful	post-experiment feedback from
Understanding of candidate patterns	Architectural Patterns	external	ordinal	patterns count	More than 10, 8 to 10, 4 to 7, 0 to 3	post-experiment feedback

Table 6.3: *List of independent variables*

3. *H12:* The use of pattern participants relationships helps software architects to better document architectural design decisions as compared to documenting design decisions without using such relationships.

   The documentation of important design decisions and rationale to support these decisions. An important aspect for applying design decisions is to ensure that major quality requirements are not compromised.

4. *H13:* The use of pattern participants relationships helps software architects to more effectively partition software architecture into components and sub-components, and assign responsibilities as compared to partitioning software architecture without the use of such relationships.

6.4.2 Variables

We used independent variables as presumed 'causes' and dependent variables as presumed 'effects' [96]. Dependent variables are not manipulated in this study and are presented as is. We use independent variables to measure their influence on the final results. For instance, a participant having a very good understanding of architectural patterns may significantly influence the results gathered from a group.

Independent Variables: We considered four independent variables prior to conducting the experiment as listed in Table 6.3. All four independent variables namely architecture design experience, pattern modeling experience, belief in using patterns, and understanding of candidate architectural patterns are measured according to ordinal scale as per the measurement scales documented in [78].

Dependent Variables: The experiment used four dependent variables, as shown in Table 6.4 for analyzing the integration of architectural patterns within software architectures. The four dependent variables correspond to the four hypothesis. We evaluate the consequences of integrating architectural patterns, once with the use of pattern participants relationships, and once without, respectively for the treatment and the control group. Each of the four dependent variables is measured according to an interval scale with values ranging from 1 to 10 (1=poorest, 10=good). The selection of the interval measurement scale was based on the measurement scales documented in [78].

Name of the variable	Class/Entity	Type of attribute (internal/ external)	Scale	Measurement Unit	Range	Counting Rule
Pattern Integration	Architectural Patterns	internal	interval	numeric	1 to 10	score
Design Comprehensibility	Software Architecture	internal	interval	numeric	1 to 10	score
Design Decisions Documentation	Architectural Patterns	internal	interval	numeric	1 to 10	score
Architecture Decomposition	Software Architecture	internal	interval	numeric	1 to 10	score

Table 6.4: *List of dependent variables*

6.4.3 Experiment Design

Each of the subjects participating in the experiment was specifically asked to integrate architectural patterns for designing software architecture. Two balanced groups with comparable skill levels were formed to serve the purpose. The assessment of subjects skills was performed by providing a pre-experiment questionnaire to students where they were asked to provide information about their architecture design experience, and educational background as further discussed in section 6.4.4. The outcome from this experiment was analyzed using statistical methods as discussed in Section 6.5.

6.4.4 Subjects

The subjects in the experiment were 36 graduate students from the Computer Science department of University of Groningen, Netherlands. All subjects were enrolled in a software patterns course and they had previously passed a Software Architecture course. This software architecture course also included designing a non-trivial system. In the software patterns course, students were instructed about the use of patterns as solutions to design problems, consideration of alternate patterns, integrating patterns, and patterns influence on quality attributes. Before assigning subjects to the control and treatment groups, we determined the background knowledge and software architecture design experience of the subjects through a pre-experiment questionnaire.

We believe that software architecture design requires a certain level of expertise. For instance, subjects must have some knowledge about architectural views (e.g. structural view [64], behavioral view [64]), architectural concerns (e.g. [97]), architectural elements (e.g. components, connectors ports [46]) etc. The skill level of subjects was assessed based on the subjects architecture design experience and educational background. This was achieved by seeking pre-experiment feedback from subjects. Among the 36 subjects participating in the experiment, there were 2 PhD students and 34 master students.

6.4.5 Objects

The subjects were provided with a Software Requirement Specification (SRS) of an industrial Warehouse management system, a list of candidate architectural patterns (such as Client-Server [5], Broker [59], Layers [5], Model-View-Controller [5]), a template to document design decisions, and a number of quality requirements. Additionally, handouts containing the description of several architectural patterns for designing distributed systems was provided to the subjects. All architectural patterns were alphabetically indexed on a separate sheet with page numbers for easy referencing for the subjects to search and read the description of patterns.

6.4.6 Instrumentation

Before the start of the experiment, both groups were given sufficient time to read the SRS document and ask questions to ensure their understanding of requirements. Additionally, the treatment group was provided the pattern participants relationships document. To guide subjects for documenting the design decisions, a simple template was provided where subjects were asked to document the decision number, a short description of the design decision and the rationale for supporting the design decision.

6.4.7 Data Collection Procedures

The experiment was performed in two sessions held at the same day. After an introduction of the system and a question/answer session, the subjects had two hours and fifteen minutes to design the architecture. Furthermore, the subjects were requested to fill out a post-questionnaire to document their feedback about the experiment, knowledge of the listed candidate architectural patterns, and issues in identifying participants of related patterns. Additionally, the treatment group was asked to document how helpful they found pattern participants relationships for integrating patterns. The architecture design document, design decisions document and post-questionnaires were collected from the subjects after the experiment.

6.4.8 Analysis Procedure

To analyse the data, we requested three expert reviewers to give their judgement upon the selected aspects of the software architectures. The external reviewers had several years of architecture design experience. One of them is an industrial practitioner for the past several years, while the other two have substantial industrial and academic experience. The information about the subjects allocation to the treatment and control groups, and treatment information was not revealed to the external reviewers. The expert reviewers were asked to provide both the grades and comments for each architectural design. For this purpose, a set of architecture evaluation criteria was provided to the reviewers to document their feedback. However, reviewers' identity and final results were not revealed to each other until the final data was collected.

To make analysis efficient, we considered it highly important that the reviewers reached consensus in their understanding of the selected architectural aspects that we used in the evaluation criteria such as pattern integration, design comprehensibility, design decisions, and decomposition. This was achieved by providing a brief description of each architectural aspect used in the evaluation criteria and asking reviewers to send us their feedback in case they disagree with the documented description of architectural aspect. This procedure was performed several days prior to conducting experiment. Only minor modifications to the architectural aspects descriptions were suggested by reviewers, which were revised accordingly.

To perform the statistical analysis of the data gathered from the expert reviewers feedback, we performed Levene's test [79] and t-test [98] to determine whether the differences in mean values calculated between the groups are significant. The Levene's test is used to check if the two groups have equal variances for the selected dependent variable. The t-test is used to measure whether the found differences are statistically significant [98]. The t-test calculates the chance that similar results will be produced when the experiment is repeated (i.e. the chance that mean values differ for control and treatment groups).

6.4.9 Validity Evaluation

We improved the reliability and validity of the experiment and data collection in two ways. First, by performing a pilot run of the experiment with one subject few days prior to conducting the experiment and taking feedback from the subject about any issues in understanding and executing the plan. This subject did not participate in the real experiment execution and he had no contact with any of the subjects participating in the experiment. Secondly, we ensured that one of the authors was available to the participants during the entire experiment, in case they faced any issues like understanding the design decisions template, availability of paper sheets etc. Furthermore, the design of the experiment was revised several times by sharing the study design with researchers having good know-how of empirical research, and changes were made where necessary.

6.5 Execution of the experiment

This section discusses the instantiation of samples, randomization, instrumentation, execution of the experiment, data collection, and validation of results.

6.5.1 Sample

- *Blind experiment:* The subjects in the experiment were not told about the hypothesis i.e. we performed a blind experiment.
- *Blind Task:* To ensure that all participants had the same knowledge of the system to be designed, the system description for which the architecture has to be designed and information about the use of pattern participants relationships etc. were kept secret until the day of exercise.
- *Technology restriction:* To make sure that the use of technology did not influence the results, students were not allowed to use software architecture design tools or refer to the internet.

6.5.2 Preparation and Data Collection

The preparation and data collection went smoothly according to the experiment design described in Section 6.4 and Section 6.4.7.

6.5.3 Validity Procedure

No major problems were encountered during the execution of the experiment. One participant was concerned with the extent to which he should document the design decisions. The subject was briefly consulted and guided appropriately.

6.5.4 Statistical Analysis of the data

With the availability of a limited number of subjects for software architecture design experiments, we believe it is important to obtain maximal information from the data gathered to draw any conclusion. The t-test and Levene's test are used to analyze the numerical data and subjective analysis is performed to interpret the results.

The t-test aims at hypothesis testing to answer questions about the mean of the data collected from two random samples of independent observations. The Levene's test is performed for equality of variances among the control and treatment groups. For the Levene's test, if the significance value is less than or equal to 0.05, then equal variances is not assumed or else the variance for both groups is considered to be equal. Separate graphs are used to present data generated from the resulting software architectures. The statistics (using demographics and tables) show the difference in the results between the control and treatment groups.

6.6 Results of the Experiment

In this section, for each dependent variable, we document the number of subjects (N), the mean(M), the standard deviation (SD), and the standard error of mean (SEM) of the samples. The t-test is performed to test if the null hypothesis can be rejected.

6.6.1 Pattern Integration

The integration of architectural patterns within a software architecture often requires new communication links, results in removing certain pattern participants or some participants being merged into a single element. The appropriate integration depends on how correctly and explicitly these tasks are performed by software architects. Table 6.5 shows the mean, standard deviation and standard mean error for the control and treatment groups. The SD value for the control group is slightly higher than the treatment group indicating less variation in the individual scores of the treatment group from the mean value. There is a significant difference between the resulting mean values for the two groups (control group = 5 and treatment group = 6.6), which shows the better performance of the treatment group for integrating architectural patterns as compared to the control group.

The Sig. value from Levene's test is greater than 0.05, as shown in Table 6.6, which shows almost equal variances among the control and treatment groups. We perform the t-test to analyze the data gathered for the 'pattern integration' aspect. Table 6.6 shows the statistics of the data. The t-test with 32 degrees of freedom(df) generates p value equal to 0.003, which is considered to be statistically significant. The p-value 0.003 shows more than 99 percent confidence that the treatment group performed better as compared to the control group.

	Group	N	M	SD	SEM
Patterns Integration	Control	16	5	1.26	0.32
	Treatment	18	6.6	1.14	0.27
design comprehensibility	Control	16	4.9	1.39	0.35
	Treatment	18	6.3	1.16	0.27
design decisions	Control	16	5.2	1.56	0.39
	Treatment	18	6.5	1.48	0.35
architecture decomposition	Control	16	4.9	1.53	0.38
	Treatment	18	6.2	1.33	0.31

Table 6.5: *Statistical results for different variables*

6.6.2 Design Comprehensibility

The completeness and clarity of the resulting software architecture adds to the comprehensibility of the software architecture. Table 6.5 shows the mean, standard deviation and standard error mean values for the 'design comprehensibility' variable.

There is a significant difference between the mean values calculated for both groups (control group = 4.9 vs. treatment group = 6.3), which provides an indication that the treatment group performed better than the control group.

As a pre-requisite to run the t-test, Levene's test is performed to check the equality of variances among both groups. The Sig. value from Levene's test is greater than 0.05, as shown in Table 6.6, which shows equal variances among the control and treatment groups [79]. We perform the t-test to analyze the data gathered after the experiment. Table 6.6 shows the statistics of the data. The p value equals 0.01, which is considered to be statistically significant. The p-value 0.01 shows the probability that 1 in 100 randomization of subjects can lead to different results. The t-test value of 2.99 indicates that the treatment group performed better as compared to the control group as documented in [98].

		Levene's test for equality of variances	t-test for equality of means			
		Sig.	t	df	p	Mean Diff.
pattern integration	equal variance assumed	0.58	3.25	32	0.003	1.6
design comprehensibility	equal variance assumed	0.56	2.99	32	0.01	1.4
design decisions	equal variance assumed	0.91	2.53	32	0.02	1.32
architecture decomposition	equal variance assumed	.61	2.31	32	0.03	1.3

Table 6.6: *T Test results for different variables*

6.6.3 Design Decisions

Architectural patterns are considered an important mean for documenting design decisions [79]. We evaluate the effectiveness of using pattern participants relationships for documenting design decisions. Table 6.5 shows the mean, standard deviations and standard error mean values for the treatment and control groups. The treatment group has scored a higher mean value as compared to the control group (treatment group = 5.2, control group = 6.5). The Levene's test and t-test are performed to verify the significance of difference in mean values.

The Sig. value from Levene's test in Table 6.6 shows equal variances among the control and treatment groups. We perform the t-test to analyze the data gathered for the 'design decisions' variable. Table 6.6 shows the statistics of the data. The p value equals 0.02, which is considered to be statistically significant [98]. The p-value 0.02 shows the probability that 2 in 100 randomization of subjects can lead to different results [98], which shows that the results are statistically significant and would allow us to reject the null hypothesis. We can be confident that the treatment group performed better as compared to the control group.

6.6.4 Architecture Decomposition

An important aspect for modeling architectural patterns is decomposition of software architecture into manageable components and sub-components. Effective integration of architectural patterns can result in well partitioned software architecture [5]. Table 6.5 shows the mean, standard deviation and standard mean error of data collected for the control and treatment groups. There is a significant difference in the resulting mean value for the two groups (control group = 4.9 and treatment group = 6.2), which shows the better performance of the treatment group for decomposing the software architecture as compared to the control group.

The Sig. value from the Levene's test is 0.61 which shows high level of homogeneity in variance between both groups [79]. We perform t-test to analyze the data gathered for the 'design

decomposition' variable. Table 6.6 shows the statistics of the data. The p value equals 0.03, which is considered to be statistically significant [98]. The p-value 0.03 indicates the probability that 3 in 100 randomization of subjects can lead to different results [98], which is statistically negligible and we can be confident that the treatment group performed better as compared to the control group. The t-test value of 2.31 indicates that the treatment group mean is greater than the control group mean as documented in [98].

6.6.5 Data set reduction

As the subjects were specifically asked to merge architectural patterns in their assignment, the exclusion criteria was based on the use of at least 4 patterns or any data point that is more than 2 standard deviations away from the mean [98]. Figure 8.14 in appendix F shows the data plot graphs for overall mean scores obtained by individual subjects with respect to the four architectural aspects considered in this study. Among 36 subjects considered in this study, the data gathered from two students was excluded from the study based on the above defined criteria and subjects inclusion/exclusion criteria documented in [21]. Appendix F further discusses the exclusion of outliers in this study.

6.6.6 Hypothesis Testing

Figure 6.15 shows the average score for both groups w.r.t the four aspects considered in this study. All four aspects considered in this study have p-values in the range 0.003 to 0.03 which are considered statistically significant to reject the null hypotheses.

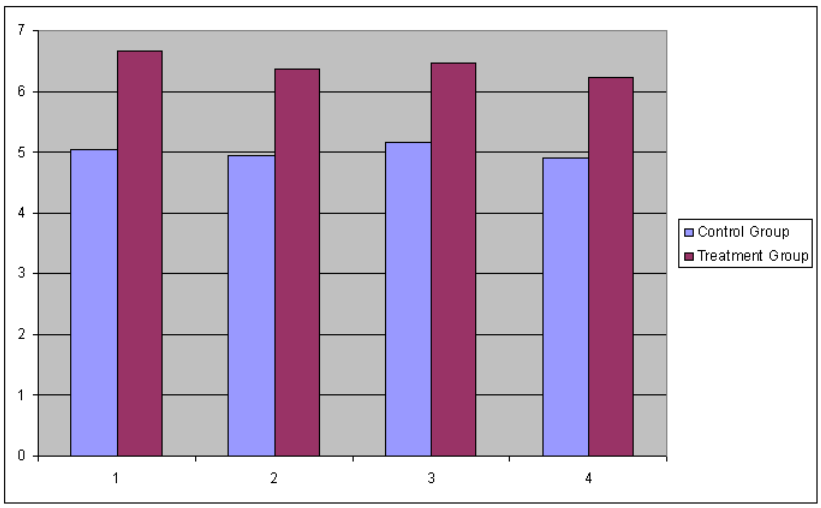

Figure 6.15: *Average scores obtained by the control and treatment groups*

While there is more than one dependent variable used to test the hypotheses, it is obvious from the results that the treatment group managed to more effectively integrate architectural patterns within software architectures, compared to the control group.

6.7 Interpretation

6.7.1 Evaluation of qualitative data and implications

We performed analysis of the qualitative data received from the expert reviewers and participants, in addition to the statistical analysis performed in the previous section. The qualitative data was gathered in two forms: feedback from the participants in the post-questionnaires, and expert reviewers feedback regarding individual software architecture documents. The analysis of the qualitative data can provide additional information to assist with the interpretation of quantitative results as presented here:

- There were two major design problems identified by the expert reviewers in the control group that were a direct consequence of 'inappropriate' integration of architectural patterns, as we concluded from the textual feedback given by reviewers. In one design document, the architect modeled patterns as 'black boxes' providing no connections among pattern participants. In another case, the architecture was considered too generic to fit the system context. In comparison, the treatment group managed to better address the design problems by coming up with more comprehensible software architectures as compared to the control group.

- By mapping the post-experiment questions about understanding of the listed candidate architectural patterns, it was noticed that the participants in the treatment group with better understanding of the listed candidate patterns managed to produce better quality architecture as compared to the participants with similar level of expertise in the control group. This leads us to a possible conclusion that pattern understanding alone is not enough to produce high quality software architecture but the effective integration of patterns improves the quality of software architecture.

6.7.2 Limitations of the Study

Threats to internal validity:

Internal validity is the degree to which the values of dependent variables can be attributed to the experiment variables e.g. balancing groups, use of statistical method, etc.

- In order to avoid bias in allocating participants to the treatment group, we assigned participants to each group randomly based on their expertise level. For instance, if there were 6 subjects with same level of expertise(i.e. experience, education background etc.), 3 were randomly picked for the treatment group and the other 3 for the control group.

- Another threat to validity is the selection of appropriate statistical method for data evaluation. We addressed the issue by sending data to an expert in the field of statistics and by studying alternate statistical methods to pick one that best fits the nature of data gathered after the experiment i.e. interval scale, scores ranging from 1 to 10 etc.

- *External reviewers bias:* There is a possibility that external reviewers may be biased in grading one or more architectural aspects considered in this study. This is because the expert reviewers may interpret a selected architectural aspect differently e.g. the design comprehensibility aspect may be interpreted differently by different reviewers. In an effort to reduce the impact of reviewers bias on final results, the selected aspects were discussed with the reviewers to seek their feedback. A brief description of the aspects was then provided to three reviewers.

Threats to external validity:

- There was a risk that the participants may have different educational background, which was not the case in our experiment. All participants had educational background in software engineering and computer science. This means that our results are more generalizable to 'people' with technical background than with a non-technical background.
- *Generalization:* The subjects who participated in the experiment (graduate students) are unlikely to be representative of experienced industrial software architects. However, Sjoberg et al. in [81], have also suggested that graduate students of computer science be considered as semiprofessionals and hence are not so far from practitioners. The experiment results encourage us to further exploit the use of pattern relationships for integrating architectural patterns in industrial experiments.
- *Time constraint* We believe that software architecture design is a lengthy and complex activity and not all of the architectural aspects (i.e. architectural views, detail component partitioning etc.) can be addressed in a limited time frame. However, subjects were asked to perform a limited task, to integrate architectural patterns within software architecture. In the design of the experiment, we considered two hours and fifteen minutes sufficient for subjects to come up with reasonable architecture. The decision to allocate 2 hrs 15 mins time slot for each group was verified by a pre-experiment pilot run of the study.
- There is a risk that the three expert reviewers may significantly differ in rewarding grades to a specific software architecture. To avert this risk, we performed inter-rater agreement test to identify major differences in grades. The inter-rater correlation test was used to identify the degree of homogeneity in grades. The Tables in Appendix G provide the results of performing the inter-rater correlation test, which shows acceptable level of difference in homogeneity of grades.

6.8 Conclusions and Future Work

This chapter presented an approach to support practitioners in the task of integrating architecture patterns by documenting a list of relationships at the level of pattern participants. The approach was validated through a controlled experiment. Four aspects were taken into consideration for integrating architectural patterns: pattern integration, design comprehensibility, design decisions, and architecture design decomposition. The subjects which were provided pattern participants relationships managed to more effectively integrate architectural patterns within software architectures as compared to participants which were not provided such information. The results from our experiment show that a more rigorous documentation of relationships among architectural patterns can help inexperienced architects to come up with higher quality software architectures. We can further make the following comments: a) understanding of architectural patterns can not guarantee by itself a good application of patterns in an architecture unless architectural patterns are effectively integrated; b) the four aspects considered in this study for analyzing the quality of software architectures are only a few of many architectural aspects, all of which require more empirical research.

6.9 Acknowledgements

We would like to thank Neil Harrison, Matthias Galster, and Simon Gieseke for reviewing the architecture documents and providing their valuable feedback to draw conclusions from this

study. We are also thankful to Peter van Saten for helping us analyze the statistical data collected after this experiment. Moreover, we thank the software patterns course graduate students from the University of Groningen, the Netherlands for their participation in the experiment.

Chapter 7

Conclusions

This chapter documents the conclusions of this thesis. First the answers to research questions are presented. This is followed by contributions this thesis makes for designing software architecture using architectural patterns. The chapter concludes with a discussion about future work and open research issues.

7.1 Research questions and answers

In this section, we will discuss how the research questions in Chapter 1 have been addressed by the chapters 2 to 6. In the introduction chapter one overall research question was formulated. We first discuss the research questions RQ1 to RQ4 and later address the overall research question.

RQ-1: ***What support the existing modeling languages offer for modeling architectural patterns?***

Our survey in Chapter 2 evaluates six modeling languages including UML for modeling four architectural patterns. An evaluation framework was used to serve this purpose. The evaluation framework consists of the following criteria:

- Syntax - expressing pattern elements, topology, constraints and configuration of components and connectors
- Visualization - graphical representation for modeling patterns
- Variability - the ability to express not only individual solutions but the entire space of solution variants
- Extensibility - capability to model new patterns

We identified that the modeling languages offer a varying support for modeling patterns. We argue that the existing modeling languages are weak in effectively expressing the solution of patterns. A language offering high extensibility support often falls behind in visualization support and another one offering good syntax support provides weak support for extensibility and so on. For each modeling language discussed in this chapter, some of the strong and weak points were highlighted for their support to model patterns.

RQ-2: ***How to effectively model the solution of a specific pattern variant?***

In Chapter 3 and Chapter 4, we study the solution of several architectural patterns in structural and behavioral views and document recurring solution structures called architectural primitives. The main idea behind the documentation of architectural primitives is that the primitives consist of small recurring solution structures, which are less likely to vary, for their use in modeling different architectural patterns. An example of the architectural primitive documented in both the structural and behavioral views is the Push-Pull primitive. Push, Pull, and Push-Pull structures occur when a target component receives a message on behalf of a source component (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull). The Push-Pull primitive is a recurring structure found in the solution of several patterns like Pipes and Filters and Publish-Subscribe. We present several architectural primitives and a vocabulary of pattern-specific architectural elements. In both chapters, we demonstrate the modeling of specific pattern variants using primitives and pattern-specific elements. We advocate that the architectural primitives can be used as the basic building blocks and when applied in combination with the pattern-specific elements can assist in modeling specific variants of an architectural pattern.

RQ-3: ***How to effectively model the solution of any variant of a pattern?***

Our contribution to model any variant of an architectural pattern is the categorization of pattern's solution participants. The categorization of the solution participant of patterns was described in Chapter 5 where we presented a pattern variant modeling approach for modeling several pattern variants. A brief summary of the categorization of patterns' solution participants is as follows:

- *Architectural Primitives:* recurring architectural solutions discovered in several architectural patterns,
- *Generic Pattern Participants:* pattern participants within the original solution specified by architectural patterns, and
- *Specialized Pattern Participants:* specializations of generic pattern participants for modeling specific pattern variants.

Using the aforementioned solution participants of patterns with the help of a modeling approach, as discussed in Chapter 5, different variants of architectural patterns can be effectively modeled. With examples and results from a controlled experiment, we illustrated the approach for modeling architectural patterns variants. We demonstrate that the use of generic and specialized pattern participants in conjunction with architectural primitives offers reusability support by providing a vocabulary of design elements that entail the properties of known pattern participants that are extensible enough to be specialized for modeling any variant of a pattern.

RQ-4: ***How to effectively integrate architectural patterns and pattern variants within software architectures?***

The main contribution of our work to address this research question is mining and documentation of 8 types of relationships between the participants of architectural patterns. These relationships have been identified by studying several industrial case studies and pattern integration examples documented in the literature. The relationships discovered by us cover different possible associations between the participants of architectural patterns, which correspond to alternative design solutions. Few examples of such relationships are:

- *Redundant pattern participants: absorbParticipant* - In an absorbParticipant relationship, certain participants of a pattern are absorbed by the participants of another pattern to avoid redundancy.

- *Overlapping pattern participants: mergeParticipant* - The mergeParticipant relationship is used to integrate one or more semantically different pattern participants into a single participant within the target pattern.

- *Modeling patterns within the participant of a target pattern: importPattern* - importPattern is a relationship where the participant(s) of a target pattern import all participants from a source pattern. This means all participants of a pattern are modeled within the participant of another pattern.

- *Modeling participants within another pattern participant: importParticipant* - An importParticipant is a relationship where participants of the target pattern import *specific* participants from the source pattern.

The approach was validated through a controlled experiment. Four aspects were taken into consideration for integrating architectural patterns: pattern integration, design comprehensibility, design decisions, and architecture design decomposition. The results from our experiment showed that a more rigorous documentation of relationships among architectural patterns can help inexperienced architects to come up with higher quality software architectures.

The answers to research questions **RQ-1** to **RQ-4** helped us answer the overall research question. The main research question is as follows:

RQ: ***How can the architects effectively model pattern variability and integrate patterns during the phase of software architecture design?***

This thesis made a contribution to address this research question. The challenge for modeling pattern variability has been addressed in Chapters 3, 4, and 5 where we used architectural primitives in combination with pattern variant-specific element for successfully modeling several different pattern variants. Moreover, using the catalogue of pattern participants relationships presented in Chapter 6, software architects can effectively integrate architecture patterns for designing software architecture.

7.2 Contributions

The research work discussed in the previous subsection lead to several possible solutions that are documented in this thesis. In general, the contribution of this thesis is to help software architects to model architectural patterns in a better way. A brief summary of the contributions is as follows:

- *Evaluation of modeling languages support for modeling patterns:* In Chapter 2, we present a set of evaluation criteria for comparing modeling languages. The evaluation results highlight the strengths and weaknesses of modeling languages for modeling patterns. It also shows that existing modeling languages are focused on designing software architecture in general with less explicit support for modeling patterns. Our work on one side allows to judge the suitability of a modeling language for pattern-based design of an architecture and on the other side leads to the conclusion that there exists a gap between architectural patterns and modeling languages. The UML-based pattern modeling solutions presented in this thesis help fill this gap.

- *Architectural primitives:* Chapter 3 and Chapter 4 document several architectural primitives in the structural and behavioral views. It helps architects take advantage of coarse-grained architectural solutions that can be used for modeling several patterns and pattern variants, and show it in architecture diagrams.
- *Pattern modeling process:* Chapter 5 presents a pattern modeling process that is aimed at modeling several variants of patterns. The work in this chapter gives an insight on how the solution of patterns can be categorized into different participants and how the use of such participants can result in better tackling the challenge for modeling pattern variability. This work also provides a mapping between pattern variants and primitives.
- *The Primus tool:* The tool discussed in Appendix B offers support to model patterns using primitives. The tool facilitates architects to visually add, modify, or delete primitives and patterns in a software architecture. The tool is implemented as an open source Eclipse IDE plug-in and provide extensions to UML 2.1 metamodel. This enables an architect to use both UML notations and architectural primitives in combination, as well as add new features such as defining new patterns. This can be done during pre-design phase (define new pattern-specific UML profiles) and during/after the design phase (change/remove primitives, patterns and architectural elements). Thus the tool provides full customization/extensibiilty support for designing software architecture using architectural patterns.
- *Understanding the relationships between patterns:* Chapter 6 provides an in-depth study of relationships between the participants of architectural patterns. The relationships are then used to successfully integrate different architectural patterns for designing software architecture. This helps architects decide the selection among alternate patterns, identify which pattern participants interact, and help overcome design issues such as inappropriate integration of patterns. The discovery of relationships is based on the study of numerous software architecture designs and pattern integration examples documented in literature.

7.3 Future work and open issues

Various issues discussed in this thesis can be a subject of future research, as we believe there are several challenges yet to be addressed. Following, we briefly discuss some of the issues that require further research:

- *Architectural views synchronization:* Designing software architecture often includes more than one architectural view that represent different architectural concerns of stakeholders. During the process of software development, different architectural views of a system need to be in sync to represent a coherent design. To synchronize the architectural views, it is necessary that the changes made in one architectural view are accordingly reflected in related architectural views which is a non-trivial task. In particular, it is challenging to trace changes to/from the behavioral views of an architecture. In this thesis, we document architectural primitives in the structural (see Chapter 3) and behavioral views (see Chapter 4). We consider the use of architectural primitives as potential future work to provide traceability among different architectural views. Architectural primitives consist of small architectural building blocks for modeling patterns that are less prone to variability and can be traced in different architectural views to keep parts of an architecture in sync. We encourage more work in this direction for architectural views synchronization using architectural primitives.

- *Source code traceability:* During the course of this thesis work, it was noticed that nearly all modeling tools in general and UML-based modeling tools in particular support high-level source code generation. An important future work can be that the changes made in the source code must be reflected in the design of the architecture. We believe that the use of architectural primitives for such support can help software architects. Architectural primitives offer coarse grained solutions that are more easily traceable to source code and vice versa. We believe that such a research work in future may yield encouraging results in providing good traceability support.

- *Tool support:* The Primus tool is a research prototype and needs significant changes for its full-fledged use to design software architecture. Especially, it is challenging to provide user friendly support to an architect for modeling any pattern variant along with the constraints. The Primus tool has been applied to small scale pattern modeling examples only. To fully validate the applicability of the Primus tool, larger case studies are necessary. This comes out with several challenges as the current visualization support of the Primus is weak to fully grasp the design of a complete architecture.

- *Other relationships between architectural patterns:* Our study in Chapter 6 mainly focused on the discovery of relationships between the participants of architectural patterns in the Component-Connector view of architecture. However, the study must be extended to existing patterns in other architectural views like interaction diagrams. We believe more relationships between the participants of architectural patterns can be discovered in different architectural views that can assist architects in more effectively integrating patterns for designing software architecture.

- *Validating patterns:* Model checking the patterns consists of testing whether all aspects of the patterns have been applied properly and whether any pattern specific rules have been broken. The OCL is used in this work to ensure that the constraints specific to the solution of individual patterns are not violated. However, understanding OCL is in itself a challenging task. An architect must be provided with effective visualization support for defining constraints specific to individual patterns. This is another direction for future research that needs to be investigated for providing effective support to validate patterns in software architecture.

Bibliography

[1] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, Volume 1, 1993.

[2] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[3] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[4] Douglas C. Schmidt Frank Buschmann, Kevlin Henney. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley Series in Software Design Patterns, 2007.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume1*. Wiley & Sons, 1996.

[6] Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[9] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. J. Wiley and Sons Ltd., 2004.

[10] Markus Voelter, Michael Kircher, and Uwe Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.

[11] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[12] Patterns and pattern languages of program. *http://hillside.net*, 2010.

[13] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Trans. Softw. Eng.*, 22(6):390–406, 1996.

[14] Nelufar Ulfat-Bunyadi, Erik Kamsties, and Klaus Pohl. Considering variability in a system family's architecture during cots evaluation. 3412:223–235, 2005.

[15] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 133–146, 2005.

[16] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[17] Hilary J. Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: what are they, and how should we teach them? *SIGCSE Bull*, 38:96–114, June 2006.

[18] Mary Shaw. What makes good research in software engineering? *STTT*, 4(1):1–7, 2002.

[19] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, pages pp. 15–24, November 1990.

[20] Samuel T. Redwine and William E. Riddle. Software technology maturation. *Proceedings of the Eighth ICSE*, pages pp. 189–200, 1985.

[21] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, 2009.

[22] R.L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491 – 506, 2002.

[23] Marvin V. Zelkowitz and Dolores Wallace. Experimental validation in software engineering. *Information and Software Technology*, 39:735–743, 1997.

[24] Joline Morrison and Joey F. George. Exploring the software engineering component in mis research. *Communications of the ACM*, 38:80–91, July 1995.

[25] Dewayne E. Perry, Susan Elliott Sim, and Steve M. Easterbrook. Case studies for software engineers. *IEEE Computer Society*, Proceedings of the 26th International Conference on Software Engineering (ICSE), 2004.

[26] D.I.K. Sjoeberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on*, 31(9):733 – 753, sept. 2005.

[27] Andreas Jedlitschka and Dietmar Pfahl. Reporting guidelines for controlled experiments in software engineering. *IEE Proceedings*, pages 95–104, 2005.

[28] Ahmad Waqas Kamal and Paris Avgeriou. An evaluation of adls on modeling patterns for software architecture design. In *4th International Workshop on Rapid Integration of Software Engineering Techniques*, 26 November 2007.

[29] Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. Modeling variants of architectural patterns. *In Proceedings of 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008)*, pages 1–23, 2008.

[30] Ahmad Waqas Kamal and Paris Avgeriou. Modeling architectural patterns' behavior using architectural primitives. *ECSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 164–179, 2008.

[31] Ahmad Waqas Kamal and Paris Avgeriou. Modeling the variability of architectural patterns. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2344–2351, New York, NY, USA, 2010. ACM.

[32] Ahmad Waqas Kamal, Paris Avegriou, and Uwe Zdun. The use of pattern participants relationships for integrating patterns: A controlled experiment. *Wiley Journal, Software: Practice and Experience*, Vol. 1:1–27, 2011.

[33] Ahmad Waqas Kamal and Paris Avgeriou. Mining relationships between the participants of architectural patterns. *4th European Conference on Software Architectures*, 2010.

[34] Mary Shaw. Some Patterns for Software Architechture. In John Vlissides, James Coplien, and Norman Kerth, editors, *Pattern Languages of Program Design, Vol 2*, pages 255–269. Reading, MA: Addison-Wesley, 1996.

[35] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT Softw. Eng. Notes*, 19(5):175–188, 1994.

[36] Nikunj R.Mehta, Nenand Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. pages –.

[37] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[38] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[39] Rober T.Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. (IEEE Software):–, 2007.

[40] Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13. IEEE Computer Society, 1997.

[41] Paris Avgeriou and Uwe Zdun. Architectural patterns revisited - a pattern language. *Technical Report*, 2005.

[42] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, Volume 6, No. 3:213–249, 1997.

[43] T. Mikkonen. Formalizing design patterns. In *Proceedings of the 20th international conference on Software engineering*, pages 115–124, Kyoto, Japan, 1998. IEEE Computer Society.

[44] Zhang Jingjun, Zhang Yang, and Li Furong. Combinatorial model and aspect-oriented extension of architecture description language. In *Information Technology: Research and Education, 2005. ITRE 2005. 3rd International Conference*, pages 277 – 281, june 2005.

[45] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Reading, MA, 1998.

[46] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[47] B. Schmerl and D. Garlan. Acmestudio: supporting style-centered architecture development. pages 704 – 705, may 2004.

[48] Architecture description languages in practice session report. (IEEE):243–246.

[49] Morgan Bjorkander and Cris Kobryn. Architecting systems with uml 2.0. *IEEE Softw.*, 20(4):57–61, 2003.

[50] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103 –112, 2001.

[51] Paul Clements. A survey of architecture description languages. (Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD' 96)):–, 1996.

[52] Architectural description languages - a technology roadmap.

[53] Robert Allen and David Garlan. Formalizing architectural connection. IEEE(16th International Conference on Software Engineering):–, 2007.

[54] Robert Allen, Rmi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. 1998.

[55] Tools for csp. *Department of Computer Science, University of Oxford*.

[56] Ralph Melton. The aesop system: A tutorial. *The ABLE Project, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213*.

[57] Validator for xml schemas. *World Wide Web Consortium*, January 2001.

[58] Robert Allen and David Garlan. A case study in architectural modeling. pages –, 2007.

[59] Douglas C. Schmidt Frank Buschmann, Kevlin Henney. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Wiley Series in Software Design Patterns, 2007.

[60] Nick Kirtley, Ahmad Waqas Kamal, and Paris Avgeriou. Developing a modeling tool using eclipse. *International Workshop on Advanced Software Development Tools and Techniques, Co-located with ECOOP 2008*, 2008.

[61] Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, New York, NY, USA, 2003. ACM.

[62] Simon Giesecke, Florian Marwede, Matthias Rohr, and Wilhelm Hasselbring. A style-based architecture modelling approach for uml 2 component diagrams. In *Proceedings of the 11th IASTED International Conference Software Engineering and Applications (SEA'2007)*, pages 530–538. ACTA Press, November 2007.

[63] Neil Harrison and Paris Avgeriou. Pattern-driven architectural partitioning: Balancing functional and non-functional requirements. In *ICDT '07: Proceedings of the Second International Conference on Digital Telecommunications*, page 21, Washington, DC, USA, 2007. IEEE Computer Society.

[64] OMG. UML 2.0 superstructure final adopted specification. Technical Report ptc/03-08-02, Object Management Group, August 2003.

[65] Uwe Zdun, Paris Avgeriou, and Carsten Hentrich Schahram Dustdar. Architecting as decision making with patterns and primitives. 2008.

[66] Object constraint language specification. *OMG Standard*, 1.1.

[67] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, Helsinki, Finland, 2003. ACM Press.

[68] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. In *Formal Methods in System Design*, pages 293–312. Kluwer Academic Publishers, 1998.

[69] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.

[70] Valentino Vranic and Jan Snirc. Integrating feature modeling into uml. In *NODe/GSEM'06*, pages 3–15, 2006.

[71] D. Kim, S. K. & Carrington. A formalism to describe design patterns based on role concepts. *Formal Aspects of Computing, Springer London*, 21:397–420, 2009.

[72] Mathias Klaus. Generic modeling using uml extensions for variability. *Intershop Research Intershop, Jena Software Engineering Group, Dresden University of Technology*, 2004.

[73] J-M Jzquel. G. Suny, A.L. Guennec. Precise modeling of design patterns. *In Proceedings of UML 2000*, volume 1939 of LNCS, Springer Verlag:pages 482–496., 2000.

[74] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in uml. In *Proceedings of the 26th International Conference on Software Engineering*, pages 252–261. IEEE Computer Society, 2004.

[75] Jing Dong and Sheng Yang. Visualizing design patterns with a uml profile. In *HCC'03*, pages 123–125, 2003.

[76] Uwe Zdun, Carsten Hentrich, and Schahram Dustdar. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Trans. Web*, 1, September 2007.

[77] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice 2nd Edition*. Addison Wesley, Reading, MA, USA, 2003.

[78] Barbara A. Kitchenham, Robert T. Hughes, and Stephen G. Linkman. Modeling software measurement data. *IEEE Trans. Softw. Eng.*, 27(9):788–804, 2001.

[79] Howard Levene. Robust tests for equality of variances. *Stanford University Press*, page pp. 278292, 1960.

[80] Brian C. Cronk. *How to Use Spss: A Step-By-Step Guide to Analysis and Interpretation*. Pyrczak Pub; 4 edition, May 2006.

[81] Jorgensen M. Sjberg D. I. K., Arisholm E. Conducting experiments on software evolution. *ACM*, Proceedings of the 4th International Workshop on Principles of Software Evolution. Vienna, Austria., 2001.

[82] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press Books. Addison-Wesley, 1995.

[83] Steve MacDonald. Design patterns in enterprise. page 25, 1996.

[84] Jordan Janeiro, Simone Diniz Junqueira Barbosa, Thomas Springer, and Alexander Schill. Enhancing user interface design patterns with design rationale structures. pages 9–16, 2009.

[85] Ronald Porter, James O. Coplien, and Tiffany Winn. Sequences as a basis for pattern language composition. *Sci. Comput. Program.*, 56(1-2):231–249, 2005.

[86] Walter Zimmer. *Relationships between design patterns*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[87] Hamza H and Fayad M. Towards a pattern language for developing stable software patterns. *Pattern languages of programming - Part 1*, 2003.

[88] M.E. Fayad and A. Altman. An introduction to software stability. *In Proceedings of Commun. ACM.*, pages 95–98., 2001.

[89] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Past, present, and future trends in software patterns. *IEEE Software*, 24:31–37, 2007.

[90] A. H. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 143–, Washington, DC, USA, 1997. IEEE Computer Society.

[91] L. Pavlic, M. Hericko, and V. Podgorelec. Improving design pattern adoption with ontology-based design pattern repository. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 649 –654, june 2008.

[92] Elspeth Golden, Bonnie E. John, and Len Bass. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. pages 460–469, 2005.

[93] Uwe Zdun Neil B. Harrison, Paris Avgeriou. Using patterns to capture architectural decisions. *IEEE Software*, pages 38–45, 2007.

[94] Grady Boosch. Handbook of software architecture: Gallery. *http://www.booch.com/architecture/architecture.jsp?part=Ga llery*, 2010.

[95] Andreas Jedlitschka and Lionel C. Briand. The role of controlled experiments working group results. pages 58–62, 2007.

[96] Evie McCrum-Gardner. Which is the correct statistical test to use? *British Journal of Oral and Maxillofacial Surgery*, 46(1):38 – 41, 2008.

[97] Schahram Dustdar and Harald Gall. Architectural concerns in distributed and mobile collaborative systems. pages 521–522, 2002.

[98] B.C. Cronk. *How to use SPSS: A step-by-step guide to analysis and interpretation*. Pyrczak Pub, Glendale, CA, 4th edition. edition, 2006.

[99] Diana L. Webber and Hassan Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.

Chapter 8

Appendices

8.1 Appendix A (relates to Chapter 3)

For the architectural primitives presented in this work, following we provide the OCL constraints used to express the semantics of architectural primitives precisely in a system design.

8.1.1 Virtual Callback

We use the following OCL constraints to define the semantics of callback primitive:

```
inv:  self.baseConnector.end.role.oclAsType(
Core::Property).class->forAll(
   c1,c2:Core::Component | c1 <> c2 implies
      c1.oclAsType(Core::Component).connects(c2))
```

8.1.2 Delegation Adaptor

To capture the semantics of Adaptor properly in UML, we use the following OCL constraints:

AdaptorPort is typed by IAdaptor as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IAdaptor.baseInterface->exists (j | j=i))
```

AdaptorPort is typed by IAdaptee as a required interface

```
inv:  self.basePort.required->size() = 1
and self.basePort.required->forAll(
   i:Core::Interface |
      IAdaptee.baseInterface->exists (j | j=i))
```

AdapteePort is typed by IAdaptee as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IAdaptee.baseInterface->exists (j | j=i))
```

AdapteePort is typed by IAdaptor as a required interface

```
inv:  self.basePort.required->size() = 1
and self.basePort.required->forAll(
   i:Core::Interface |
```

```
        IAdaptor.baseInterface->exists (j | j=i))
```

Adaptor component attaches the AdaptorPort.

```
inv:  self.baseComponent.ownedPort.name = 'AdaptorPort'
```

8.1.3 Passive Element

To capture the semantics of Passive Element properly in UML, we use the following OCL constraints:

PassivePort provides the IPassive interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IPush.baseInterface->exists (j | j=i))
```

PElement attaches the PassivePort

inv: self.baseComponent.ownedPort.name = 'PassivePort'

8.1.4 Interceder

To capture the semantics of Interceder primitive properly in UML, we use the following OCL code:

A IncdrPort is typed by IRIncdr as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IRIncdr.baseInterface->exists (j | j=i))
```

A IncdrPort is typed by IFIncdr as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      IFIncdr.baseInterface->exists (j | j=i))
```

An Interceder component owns IncdrPort

```
inv:  self.baseComponent.ownedPort.name = 'IncdrPort'
```

8.2 Appendix B (relates to Chapter 3)

8.2.1 The Primus Tool

The Primus tool has been developed to provide a practical implementation of architectural primitives. The tool interacts with the UML component diagram by allowing the user to add primitives to the model and for model checking capabilities. The functionality and usage of the tool are described in the remainder of this paragraph.

Modeling a primitive using the Primus

The Primus tool is mainly intended to assist software designers in systematically modeling architectural primitives in system design. The two main areas of functionality of Primus are: 1) modeling primitives and 2) model checking of primitives. The tool allows for primitives to be applied to existing UML elements in the Component-Connector view or for primitives to be applied with all the necessary UML elements. For example, a Callback primitive consists of two components that are connected. It is possible to apply the Callback primitive to two existing components or to create new components automatically. This option allows multiple primitives to be applied to the same component. Adding primitives is achieved via a wizard whereby the wizard is opened via a context menu. A wizard offers the user with a step-by-step explanation of the possible choices and thus requires a smaller learning curve for the user.

Figure 8.1 and figure 8.2 show two images from Primus. Figure 8.1 shows some of the options from the primitive modeling wizard and figure 8.2 shows the result for modeling the Callback primitive.

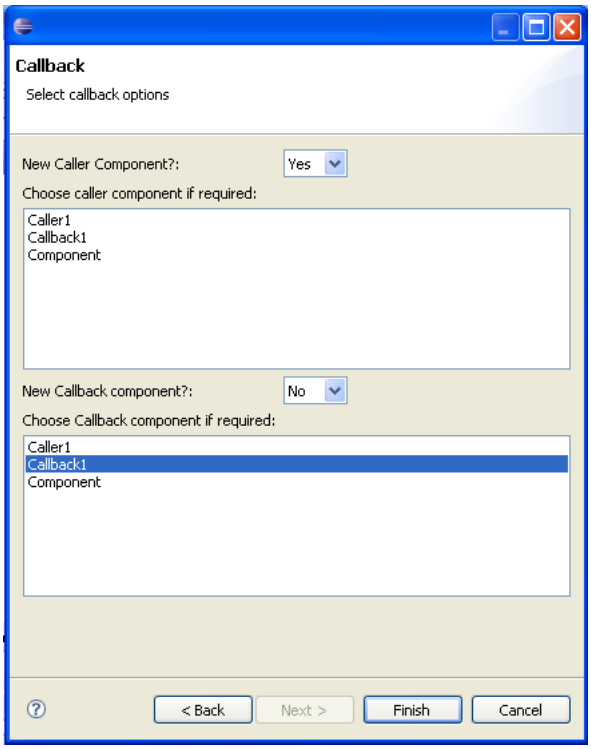

Figure 8.1: *Modeling the Callback primitive*

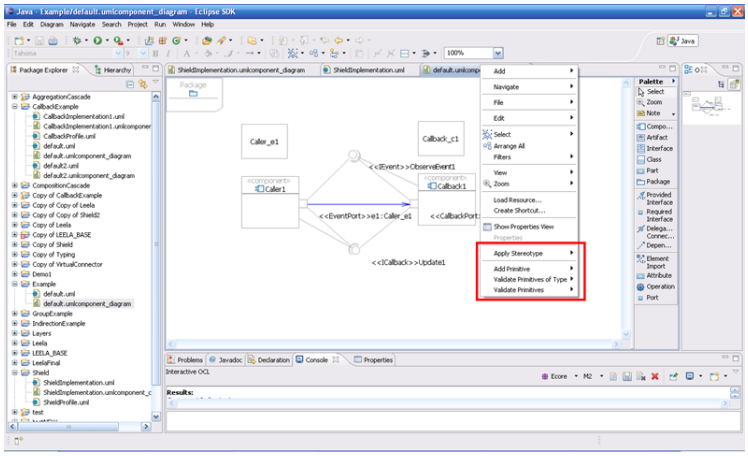

Figure 8.2: *Options for modeling primitives in Primus*

8.2.2 Pattern variants representation and validation within software architecture

Two important aspects for modeling architectural pattern variants are their explicit representation within software architecture and checking that the pattern solution is correctly applied to software architectures. The former helps better understand the architecture and reason about the quality requirements [5] while the latter ensures that the constraints specific to the solution of architectural patterns are not violated in the resulting architecture [29] e.g. the Layers pattern restricts individual layers from bypassing adjacent layers. These two aspects, as mentioned in the Introduction section, are further discussed hereafter to demonstrate how the proposed approach tackle these issues.

Explicit representation of pattern variants using primitives

The mapping of primitives to pattern variants helps to identify the pattern variants applied to a software architecture. For instance, the use of the Layering primitive hints at the presence of the Layers pattern, and Indirection hints at the presence of either the Broker, Proxy, or Message Redirector pattern. The use of the primitives in combination with the specialized pattern participants further assists in identifying specific pattern variants within software architecture. For instance, as shown in Figure 8.3, the use of the Push and Pull primitives marks the presence of data flow streams in the Pipes and Filters pattern. The explicit identification of patterns within software architectures helps to better understand the architecture and reason about the quality requirements deemed on the resulting software architecture.

Validating the primitives and pattern variants within software architecture

Model checking the primitives and pattern variants consists of testing whether all aspects of the primitives and pattern variants have been applied properly and whether any primitive/pattern specific rules have been broken. The OCL is used in this work to ensure that the constraints specific to the solution of individual pattern variant are not violated. The generic pattern participants define a generic pattern solution that is often common among all pattern variants of the same pattern. The UML's inheritance relationship supports that constraints defined for the generic participants are used in the specialized participants as well. This ensures that an ar-

chitect does not need to redefine constraints for each new variant of a selected pattern. For instance, the constraints that a Filter attaches input and output ports for receiving and sending data are defined on the generic Filter participant which stay true for all specialized Filter participants as well.

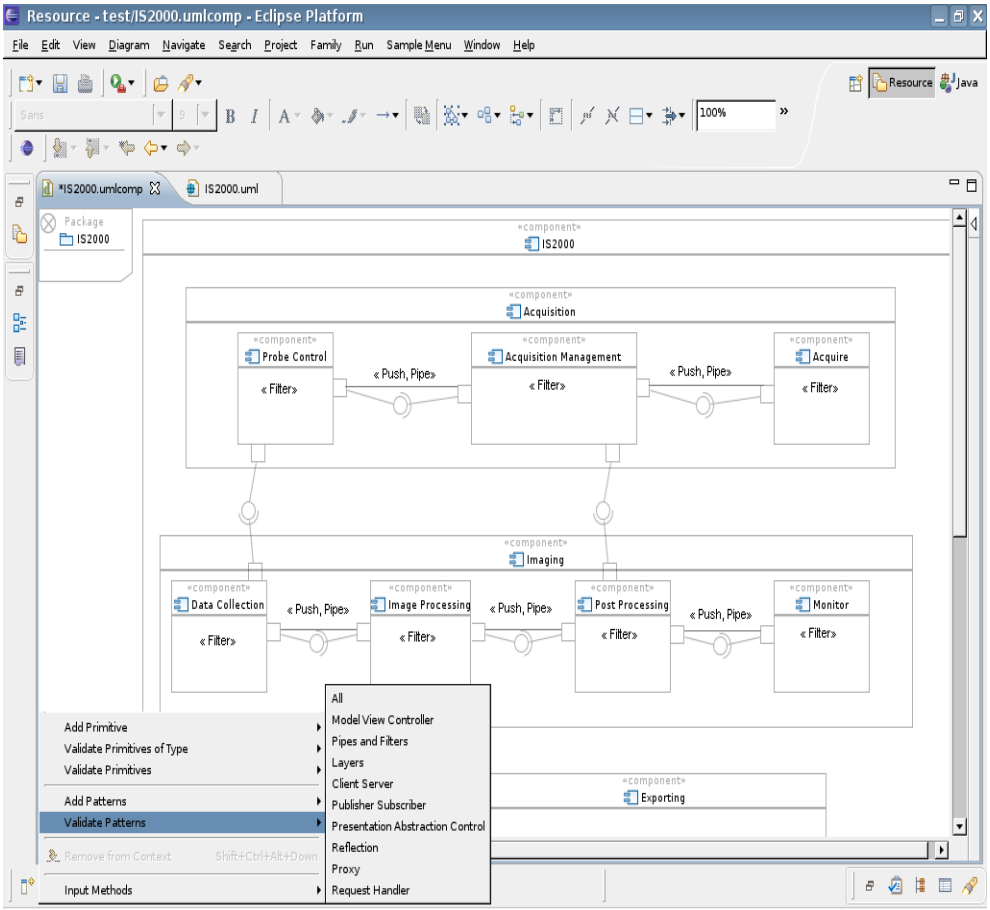

Figure 8.3: *Tool Support for Expressing and Model Checking Pattern Variants*

We will also present a validation example for the Callback primitive. If the Callback primitive has been implemented properly, the feedback to the user will something similar to Figure 1. This means that the model-checking feature has found a Callback and that it has been found between the components stated. If we now change a critical part of the Callback then it should also inform the user of the exact nature of the problem. For example, an essential part of the Callback primitive is the stereotyping of one of the interfaces with the IEvent stereotype. If we undo the application of the stereotype the model checker will return the message from Figure 5.

This message shows that the model checker is able to find a primitive that has not been applied properly and is able to state the exact problem. Therefore the user can easily correct the problem.

Using OCL to achieve the goals stated above has been achieved by returning every sub-result needed to validate a primitive. If a sub-result differs in value from the expected value then an error has been found and we automatically know what the error is. A further interpretation of the results is needed in Java so that the result can be presented properly to the user.

$$
\begin{aligned}
&\quad Component.allInstances() -> collect(i| \\
&i.ownedConnector -> collect(conn| \\
&let callerPortSter : Port = conn.end.partWithPort -> \\
&any(owner = i).oclAsType(Port) -> any(p| \\
&p.oclAsType(Port).getAppliedStereotypes() -> \\
&\quad any(name =' EventPort') -> notEmpty()), \\
&\quad callbackPortSter : Port \\
&\quad = conn.end.partWithPort -> \\
&\quad any(owner <> i).oclAsType(Port) -> \\
&\qquad any(p|p.oclAsType(Port).getAppliedStereotypes() -> \\
&\qquad any(name =' CallbackPort') -> notEmpty()), \\
&\qquad\quad Tuple(c1 = i, c2 = otherComp, callerPort = callerPortSter, \\
&\qquad\qquad callbackPort = callbackPortSter)
\end{aligned}
$$

The code above shows a segment of the Callback query. The callerportSter and callbackPortSter are sub-results that we are interested in and if a null value is returned we know that the relevant port was not stereotyped. This query will of course return many potential Callbacks within the model. The query can then be filtered to only include potential Callbacks that have x or less amount of problems. Figure 8.3 shows the GUI snapshot for pattern variants modeling and validation using the tool.

8.3 Appendix C (relates to Chapter 5)

Fig 8.4 and Fig 8.5 show the final grades assigned to individual architectures on a scale 1 to 10. The final grades are calculated as average score for the two variables used in this work.

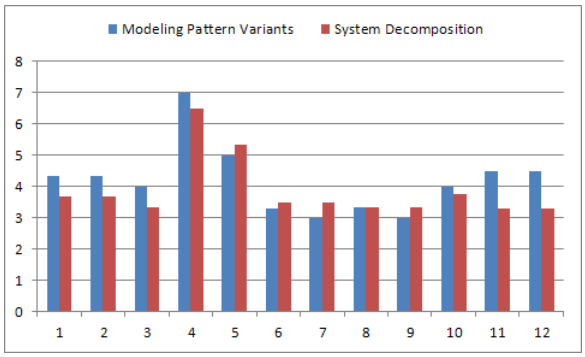

Figure 8.4: *Final score of the participants in the control group (Average Score)*

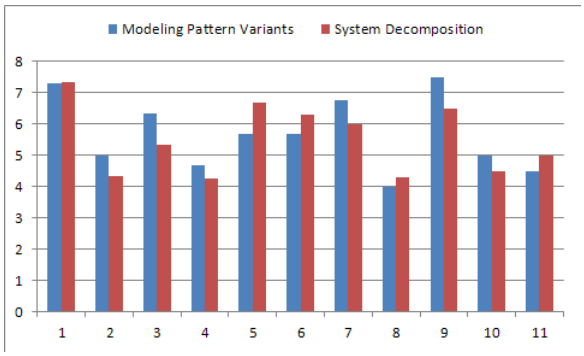

Figure 8.5: *Final score of the participants in the treatment group (Average Score)*

8.4 Appendix D (relates to Chapter 5)

The agreement among the external reviewers in assigning the grades to individual architectural aspects is calculated according to the Intraclass Correlation Coefficient as shown in Figure 8.15 (cases validity) and Figure 8.7 (intraclass correlation). The combination of agreement/dissagreement between the external reviewers is performed according to SPSS tool guidelines [80].

Case Processing Summary

			N	%
Cases	Valid		23	100,0
	Excluded		0	,0

Figure 8.6: *Case processing summary*

The results of the intraclass correlation analysis are correlation = 0.9 with Cronbach's alpha value 0.89. This measure of agreement is considered statistically significant. The intraclass coefficient value greater than 0.8 (on a scale 0 to 1) is considered significant to claim a good level of agreement [80].

Intraclass Correlation Coefficient

				95% Confidence Interval	
	Intraclass Correlation	Cronbach's Alpha	N of Reviewers	Lower Bound	Upper Bound
Measures	,900	,899	3	,801	,954

Figure 8.7: *Reliability statistics and intraclass correlation results*

8.5 Appendix E (relates to Chapter 5)

In this section, we use the UML to describe the approach presented in Chapter 5 for modeling pattern variants in UML's Component-Connector view. Figure 8.8 shows the general relationships among these concepts in UML. The UML's extension mechanism of stereotypes, constraints, and tagged values is used to express these notions. Defining architectural primitives using UML is already covered in our previous work [15] [29] while in this section we focus on defining the mechanism to express generic and specialized pattern participants. We extend the

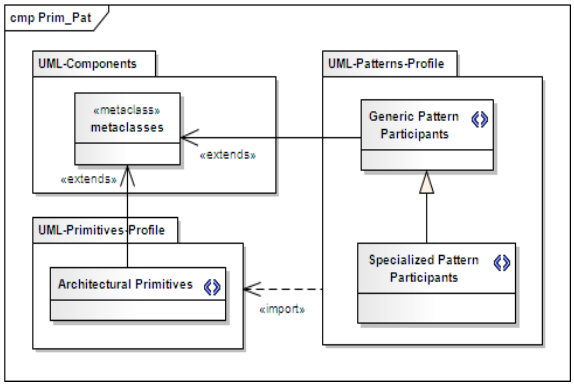

Figure 8.8: *The Relationship between Primitives, Generic and Specialized Participants in UML*

UML meta-classes in the Component-Connector view to express generic and specialized pattern participants while the tag values are used to mark the variation points and variants. The pattern participants marked as variation points represent the variation that an element entails for its use in several variants of the same pattern. For instance, in the Model-View-Controller pattern, the Model component marked as variation point is specialized as Document to express the Document-View pattern variant as explained in next section. To serve this purpose, the meta-classes Component, Connector, Port, and Interface are used as described below:

- *Expressing Generic Pattern Participants as Variation Points in UML:* We extend the above mentioned UML meta-classes to express the generic pattern participants as stereotypes. The UML's profile mechanism is used to serve the purpose. For instance, the Pipe and Filter participants of the Pipes and Filters pattern are expressed as stereotypes by extending the Connector and Component meta-classes respectively. To mark selected pattern participants as variation points, the tagged values are defined for pattern participants as a string variable. The variation in UML elements is designated by small dot symbols in UML diagrams used in this work. Although this symbol is not included in UML standard, it has been widely used in literature to denote variation points [99]. We use UML tag syntax *vp ¡VariationCategory¿ ¡Variation1, Variation2, ... VariationN¿* to show different variable choices in applying a pattern to a system design.

- *Expressing Specialized Pattern Participants in UML:* We use the UML's inheritance relationship to instantiate several variants from the generic pattern variant participants. For instance, the Feedback and Fork are the specialized pattern participants within the Pipes and Filters pattern that are expressed using the UML's inheritance relationship. Pattern participants that often work in conjunction to model a pattern in system design are expressed using UML's dependency relationship. For example, the Model participant within the MVC pattern has a dependency relationship with data or event ports to communicate

with surrounding elements. Furthermore, the same tagged values defined for expressing variation points are overridden to mark the specialized pattern participants.

We also use the following UML metaclasses in order to express the OCL constraints while traversing the UML metamodel: AggregationKind, Classifier, ConnectableElement.

Following, we demonstrate the use of pattern modeling approach documented in Section 5.4 for modeling two known variants of the Pipes and Filters pattern using UML. The approach presented below can be followed to model any pattern variant using any software modeling language as long as the selected modeling language provides enough extensibility mechanism to express patterns variants and primitives.

8.5.1 Example 1: Defining and modeling the variants of pipes and filters pattern in UML

Defining generic pattern participants of the pipes and filters pattern

To model different variants of the Pipes and Filters pattern, the generic pattern participants are defined and next, for each selected Pipes and Filters variant, the generic pattern participants are used to define specialized pattern participants. In the case of Pipes and Filters pattern, the Pipe, Filter, Input and Output are considered generic participants as these contribute, often after further specialization, for modeling several variants. Figure 8.9 shows UML profile for defining pattern participants of Pipes and Filters pattern variants. UML relationships are used to specialize, associate and define the pattern participants of selected pattern variants. Following, we first define the generic participants of the Pipes and Filters patterns and in the subsequent subsections demonstrate the modeling of two variants using primitives and pattern participants.

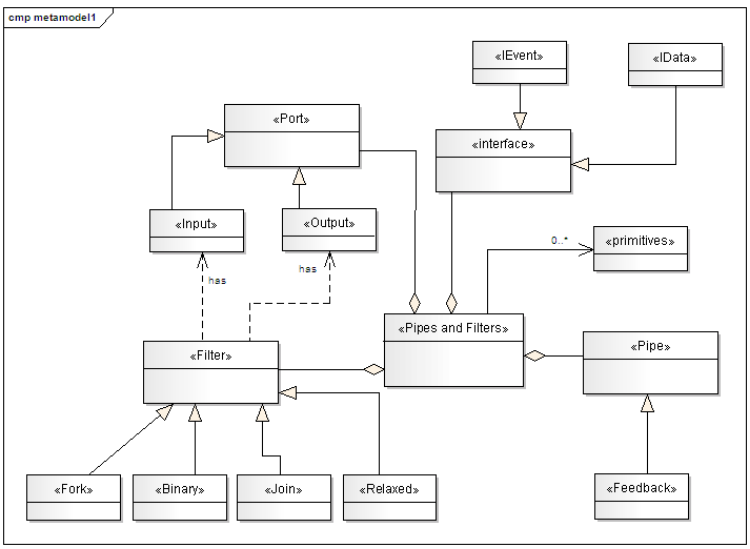

Figure 8.9: *Defining Pipes and Filters Pattern Participants*

≪*Filter*≫*:* A stereotype that extends the Component metaclass of UML and attaches input and output ports. A Filter component is formalized using following OCL constraints:

```
inv:  self.ownedPort->name() = Input | Output
```

≪*Pipe*≫ A stereotype that extends the Connector metaclass of UML and connects the output port of one component to the input port of another component. A Pipe is formalized using following OCL constraints:

A Pipe has only two ends.

```
inv:  self.baseConnector.end->size() = 2
```

≪*Input*≫ A stereotype that extends the Port metaclass of UML.

An Input port is typed by Iinput as a provided interface

```
inv:  self.basePort.provided->size() = 1
and self.basePort.provided->forAll(
   i:Core::Interface |
      Iinput.baseInterface->exists(j | j = i))
```

≪*Output*≫A stereotype that extends the Port metaclass of UML.

An Output port is typed by Ioutput as a required interface

```
inv:  self.basePort.required->size() = 1
   and self.basePort.provided->forAll(
      i:Core::Interface |
         Ioutput.baseInterface->exists(j | j = i))
```

Expressing the forked pipes and filters pattern variant

The single-input single-output structure of the Pipes and Filters pattern can vary to allow filters with more than one input/output. Such a structure can then be setup as a Forked Pipes and Filters pattern that can even contain Joins and Feedback loops.

As a first step, according the pattern variants modeling approach documented in Section 3, the primitives that participate for modeling the Forked Pipes and Filters pattern variant are selected. The *Push*, *Pull* and *Adapter* primitives are selected. The rationale for selected these primitives is as follows:

- Filters either Push or Pull data from/to the adjacent filters in the chain.
- Data is adapted according the appropriate date format before sending it to the Sink which is expressed using the Adapter primitive.

Next, the Fork and Feedback structures are missing solution aspects that need to be expressed by specializing the generic Pipes and Filters pattern participants as follows:

≪*Fork*≫ The Fork participant specializes the Filter participant of the Pipes and Filters pattern as shown in Figure 8.9.

≪*Feedback*≫ A stereotype that specializes the Pipe participant of the Pipes and Filters pattern. Figure 8.9 shows the use of UML generalization relationships for defining the Feedback participant.

≪*Join*≫ A *Join* specializes the Filter participant of the Pipes and Filters pattern as shown in Figure 8.9 and enforces additional constraints as defined below:

A Join participant attaches input port which is typed by two required interfaces of different pipes, as constrained using following OCL code:

```
inv:  self.baseComponent.ownedPort() = ?input' and
self.basePort.required->forAll(
```

```
i:Core::Interface |
   Iinput.baseInterface->size() = 2)
```

A combined use of specialized participants and primitives yield the Forked Pipes and Filters pattern variant with Feedback loop as shown in Figure 8.10.

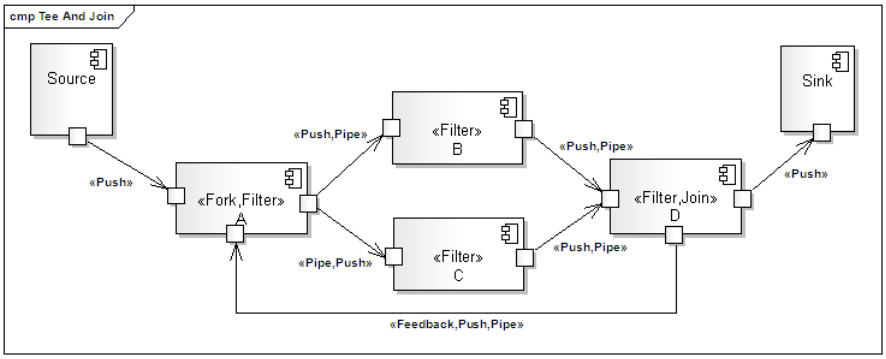

Figure 8.10: *Modeling the Forked Pipes and Filters Pattern Variant*

Expressing the passive parallel processing pipes and filters pattern variant

Parallelism and distributivity for processing streams of data is a preferable option to design real time distributed systems [5]. To improve efficiency and scalability of such a system, the data with different nature can be separately processed using different pipelines. For parallel processing, such a model uses Forks and may include Passive Filters as buffers to ensure that all processed data is delivered to the sink.

According to the pattern modeling approach documented in section 5.4, the primitives that participate for modeling the parallel processing Pipes and Filters pattern variant are selected. The Passive Element, Push, and Pull primitives can be used to express the parallel processing Pipes and Filters pattern variant. The rationale for selecting these primitives is as follows:

- The Push and Pull primitives are used to send/receive data to adjacent filters in the chain.
- The Passive Element primitive is used to mark the passive filters in the chain.

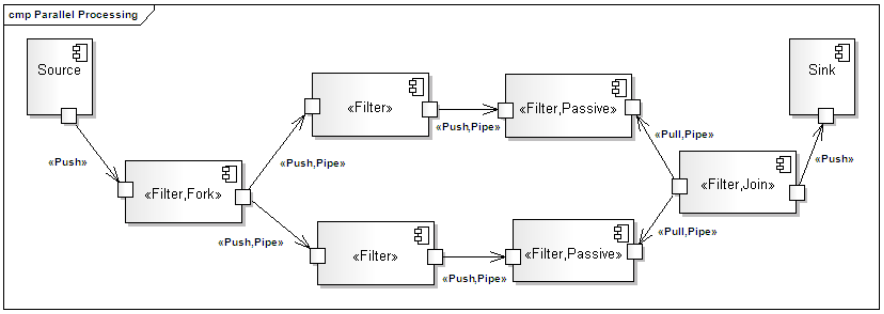

Figure 8.11: *Parallel Processing Pipes and Filters Pattern Variant*

After the selection of suitable architectural primitives, as a next step, the generic participants of the Pipes and Filters are considered. The generic participants fall short in expressing

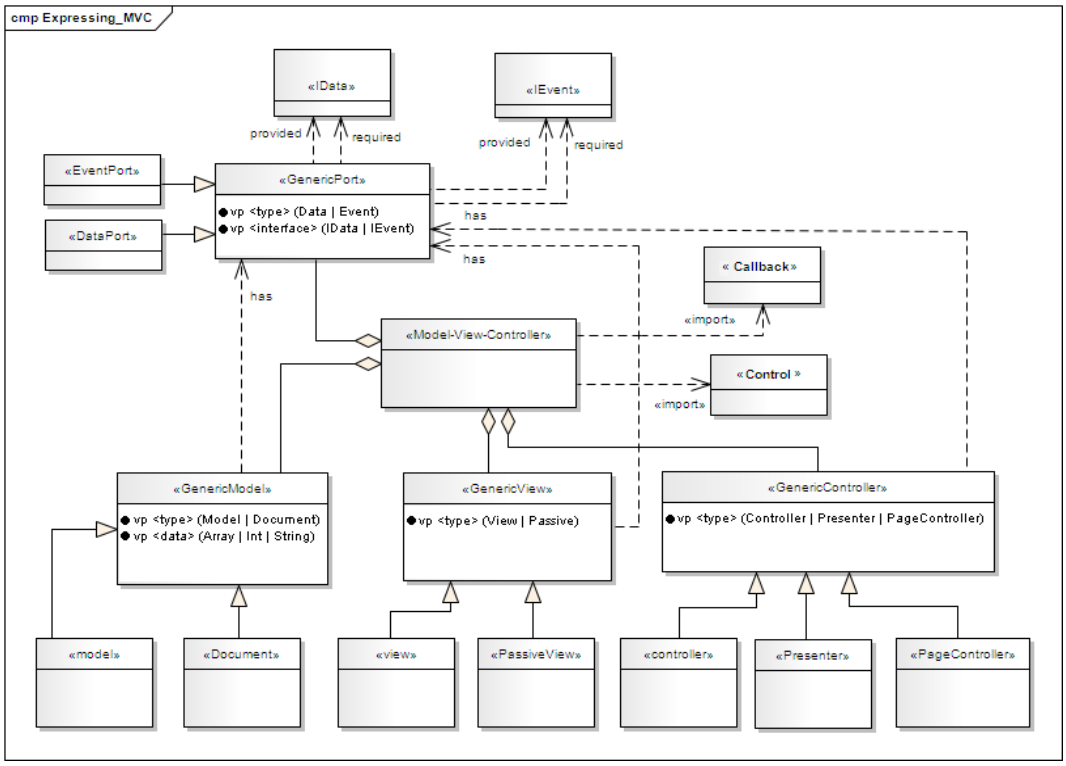

Figure 8.12: *The Model-View-Controller defined in UML*

the solution of passive parallel processing pipes and filters pattern. The Fork is the missing solution aspect, however, it is defined in the previous subsection as part of the UML profile. The primitives and specialized pattern participants are applied to UML components and connectors for expressing the Forks Pipes and Filters variant as shown in Figure 8.11.

8.5.2 Defining and modeling the variants of Model-View-Controller Pattern in UML

The structure of the MVC pattern consists of three components namely the Model, View, and Controller [5]. The Model provides functional core of the application and updates views about the data change. Views retrieve information from the Model and display it to the user. Controllers translate events into requests to perform operations on View and Model elements. In such a structure, the Model component provides services to the View and Controller components. Following we use the approach presented in section2 to define participants of the MVC pattern and its variants in UML as shown in Figure 2.

In the solution specified by MVC pattern, the View subscribes to the Model to be called back when some data change occurs. Such a structure can be effectively expressed using the *Callback* primitive. Also, the Controller sends events to the Model for an action to take place, which can be expressed using the *Control* primitive.

The *Callback* and *Control* primitives express part of the solution specified by the MVC pattern. The Model, View and Controller are the generic pattern participants within the MVC pattern that lead to several different forms within individual pattern variants hence marked as variation points. The participants of the MVC pattern use both the event and data based services realized using the ports attached to the Model, View, and Controller components which

are used to send/receive data or events. In the specific case of the MVC pattern, the variability in communication is covered by the *Callback* and *Control* primitives and hence not marked in the MVC profile shown in Figure 2.

GenericModel: A stereotype that extends the Component meta-class of UML and attaches ports for interaction with the Controller and View components that is formalized using the following OCL constraints:

```
   Component.allInstances()->iterate(
   i;pairs : Set(Tuple(c1 : Component,
      s : Bag(Component))) = Set |
         let comp : Component = i.oclAsType(Component),
            stemp : Bag(Component) =
   comp.ownedConnector->select(j |
      let Callback : Port = j.oclAsType(
         Connector).end.partWithPort->any(
            owner=i).oclAsType(Port),
               EventPort : Port =          j.oclAsType(
         Connector).end.partWithPort->any(
            owner<>i).oclAsType(Port) in
if
   j.oclAsType(Connector).getAppliedStereotypes()->
      any( name='Callback ')->notEmpty() and
      EventPort.getOwner() = 'GenericView'and
      Model.ElementType = 'vp'
then
   true
else
   false
endif
```

GenericController: The controller stereotype is an extension to the Component metaclass of UML and attaches ports for interaction with the Model and View components. The controller is formalized using the following OCL constraints:

```
   let comp : Component = i.oclAsType(Component),
   stemp : Bag(Component) =
      comp.ownedPort->select(j |
         j.oclAsType(Port), self.ownedPort =
         EventPort and i.oclAsType(Port) in
if
   EventPort.getOwner() = 'GenericView 'and
      Controller.ElementType = 'vp '
then
   true
else
   false
endif
```

GenericView: A stereotype that extends the Component meta-class of UML and attaches ports for interaction with the Controller and Model components which is formalized using the OCL constraints as follows:

```
   let comp : Component = i.oclAsType(Component),
   stemp : Bag(Component) = comp.ownedPort->select(j |
```

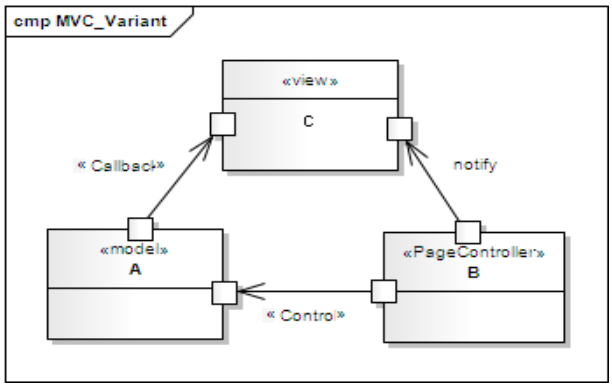

Figure 8.13: *Modeling the PageController MVC Pattern Variant*

```
      j.oclAsType(Port), self.ownedPort =
         EventPort and i.oclAsType(Port) in
if
   EventPort.getOwner() = ‘GenericModel ’and
      View.ElementType = ‘vp ’
then
   true
else
   false
endif
```

A combined use of the specialized pattern participants and primitives results in the MVC structure as shown in figure 3:

8.6 Appendix F (relates to Chapter 6)

Fig 8.14 shows the final grades assigned to individual architectures on a scale 1 to 10. The final grades are calculated as average score for four variables used in this work. Grade 2.1 from the control group and grade 2.3 from the treatment group are considered outliers and hence are not considered when performing statistical analysis in this work.

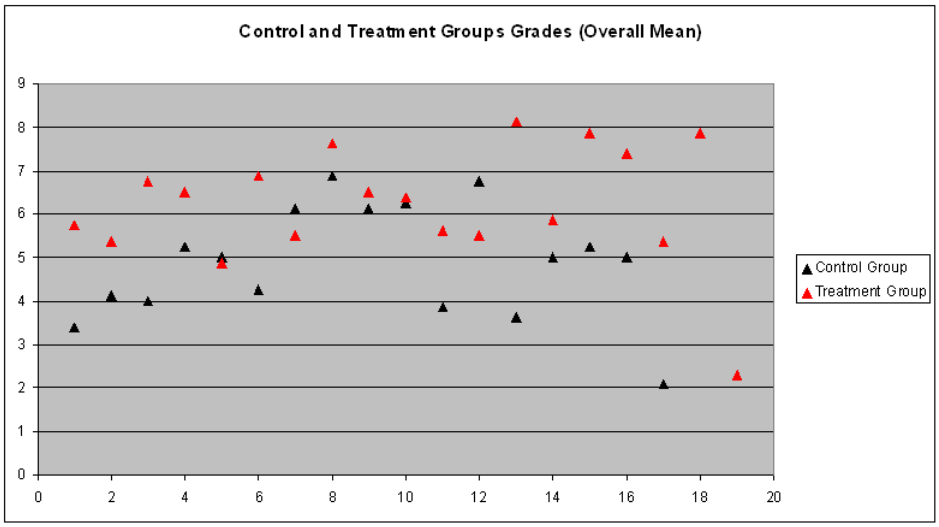

Figure 8.14: *Identification of outliers in the control and treatment groups (Average Score)*

8.7 Appendix G (relates to Chapter 6)

The agreement among the external reviewers in assigning the grades to individual architectural aspects is calculated according to the kappa statistics as shown in Figure 8.15 (cases validity), Figure 8.16 (different combinations of agreement/disagreement between reviewers), and Figure 8.17 (kappa value). The combination of agreement/dissagreement between the external reviewers, as shown in Figure 8.16, is performed according to SPSS tool guidelines [98]. For instance, the value of 18 in Figure 8.16 shows the total number of instances where reviewers had consensus in assigning a low grade to an architecture.

	Cases					
	Valid		Missing		Total	
	N	Percent	N	Percent	N	Percent
Case1 * Case2	124	100.0%	0	.0%	124	100.0%

Figure 8.15: *Case processing summary*

The results of the interrater analysis are Kappa = 0.7 with approx. sig. value less than 0.001. This measure of agreement is considered statistically significant. The Kappa values of at least 0.6 and preferably higher than 0.7 are considered significant before claiming a good level of agreement [98].

		Case2			
		1	2	3	Total
Case1	1	18	5	0	23
	2	3	51	5	59
	3	3	7	32	42
	Total	24	63	37	124

Figure 8.16: *Cross tabulation data for matching and non-matching cases*

		Value	Asymp. Std. Error[a]	Approx. T[b]	Approx. Sig.
Measure of Agreement	Kappa	.701	.056	10.685	.000
	N of Valid Cases	124			

Figure 8.17: *Symmetric measures - kappa statistics*

8.8 Appendix H (relates to Chapter 6)

In this section, we present an example to design part of a Warehouse management system [59]. A key requirement for the development of such a system is the communication middleware that offers business process management. The goal of the communication middleware is to simplify application development by providing uniform view of network services and separate core application functionality from communication complexities such as connection management, data transfer, even and request demultiplexing, and concurrency control etc. Some of the key nonfunctional requirements deemed in the resulting software architecture are scalability, portability, flexibility, and distribution. It must be noted that a software architecture design activity involves several steps like requirement analysis, prioritization of key drivers, selection of appropriate patterns, verification, and validation etc. For the sake of simplicity, we do not document all architecture design activity steps and focus only on the pattern integration process alongside the major design decisions. Also, the Warehouse management system is a large scale system and in this section we present only a part of the architecture to demonstrate the working of pattern participants relationships.

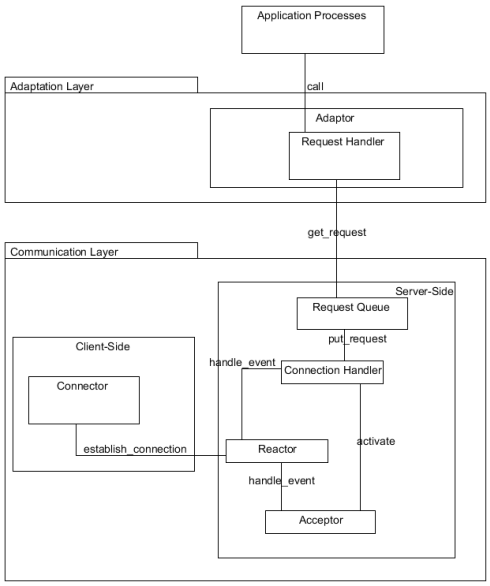

Figure 8.18: *Example Architecture Design*

We selected the Layers, Client-Server, Reactor, Acceptor-Connector, and Request Handler patterns. We considered these patterns suitable to effectively address the scalability, portability, flexibility and distribution requirements. As a first step, the communication middleware is implemented as a layered structure: the adaptation layer and the communication layer. When integrating the Reactor and Acceptor-Connector patterns, only the Handler participant from the Reactor pattern will be used to handle events. Such an integration between the Reactor and Acceptor-Connector patterns is done using the *absorbParticipant* relationship as discussed in section 6.3.3. In the core communication layer implementation, a component initiates an event loop using the Reactor pattern. When a request event occurs, the Reactor demultiplexes the request to the appropriate event handler. The Reactor then calls the handle event method on the Connection Handler, which reads the request and passes it to Adaptation layer. The *importParticipant* relationship is used to import the participants of the Reactor and Acceptor-Connector

patterns into the participants of the Client-Server pattern. Moreover, the Adaptation layer imports the Adaptor pattern using *importPattern* relationship. This layer then demultiplexes the request to the appropriate *call* method. Figure 8.18 shows the resulting software architecture for integrating the above listed architectural patterns using pattern participants relationships.