

The Concern-Oriented Software Architecture Analysis Method

Author:	Frank Scholten
E-mail:	f.b.scholten@cs.utwente.nl
Student number:	s0002550
Supervisor:	Dr. ir. Bedir Tekinerdoğan
Graduation committee members:	Prof. Dr. ir. Mehmet Akşit Dr. ir. Bedir Tekinerdoğan Dipl.-Ing. Christian Hofmann

Abstract

Software architecture evaluation methods aim to predict the quality of a software system before it is built. Depending on the results of these evaluations, the software architecture can be transformed to improve its quality. In general, processes of transformation of software architecture designs are implicitly defined in existing software architecture evaluation methods. Additionally, these methods use scenarios and do not treat concerns as first-class entities. The explicit representation of concerns enables explicit reasoning and identification of crosscutting concerns. Crosscutting concerns may cause scattering and tangling of code fragments, which have a negative impact on the maintainability and complexity of software implementations.

The Concern-Oriented Software Architecture Analysis Method (COSAAM) is an iterative method for evaluating and transforming software architectures. COSAAM uses and extends various existing approaches to create a method to systematically transform software architectures. COSAAM inspired by the Aspectual Software Architecture Analysis Method (ASAAM), which is a scenario-based analysis method for identifying architectural aspects. Additionally, it uses Design Structure Matrices (DSMs) for concern identification and dependency analysis of architectural modules and Domain Mapping Matrices (DMMs) to measure scattering and tangling. DSMs and DMMs are general purpose system analysis tools that have been applied in project management, product design and more recently, software architecture design.

COSAAM provides transformation rules to systematically transform a candidate software architecture. COSAAM is demonstrated by a case study of a window manager software architecture. The window manager software architecture is transformed in eight iterations and is extended with several aspects. After the end of the evaluation, the scattering and tangling in the window manager software architecture is eliminated. However, this is offset with added complexity due to interacting aspects. This thesis is concluded with an overview of lessons learned during the development of COSAAM and provides suggestions for further work and tool support.

Acknowledgments

I would like to thank the members of my graduation committee, Bedir Tekinerdoğan, Mehmet Akşit and Christian Hoffman for their time and comments they have given to during my graduation project. I would especially like to thank Bedir, who has been my supervisor on several occasions, during the development of ASAAM-T, my internship and my graduation. During many meetings I have learned a lot from him about many things in software engineering.

My parents have always been very supportive in everything I have done, especially during in my final year at University. I want thank them for everything. Additionally, I want to acknowledge my best friends Rikke, Simon and Rik. We always have a great time whatever we do and wherever we go. I also want to acknowledge my housemates at Calslaan 12 for the good times in all the years living on campus.

Contents

1	Introduction	3
1.1	Software Architecture Evaluation	3
1.2	Problem Statement	3
1.3	The Design Structure Matrix and Domain Mapping Matrix . . .	4
1.4	Contributions	4
1.5	Outline of this thesis	5
2	Problem Statement	6
2.1	Software Architecture Evaluation Methods	6
2.1.1	Introduction	6
2.1.2	SAAM	7
2.2	Separation of Concerns and Crosscutting Concerns	8
2.2.1	Aspectual Software Architecture Analysis Method	8
2.3	Window Manager Case Study	9
2.3.1	ASAAM Example	10
2.4	Problems with existing software architecture evaluation methods	12
2.5	Requirements for a new evaluation method	13
3	Design Structure Matrices and Domain Mapping Matrices	15
3.1	Design Structure Matrices	15
3.2	Domain Mapping Matrices	16
3.3	DSMs and Software Architecture Design	17
3.4	DSMs and Aspect-Oriented Software Architecture Development .	18
4	The Concern-Oriented Software Architecture Analysis Method	19
4.1	Overview of COSAAM	19
4.2	Preparation Phase Overview	22
4.2.1	Activity: Define Concerns	23
4.2.2	Activity: Describe Candidate Software Architecture	31
5	COSAAM Analysis Phase	32
5.1	Analysis Phase Overview	32
5.1.1	Activity: Initialize Concern-Module DMM	33
5.1.2	Activity: Characterize Concern and Module Mapping	35

5.1.3	Activity: Measure Scattering and Tangling	50
5.1.4	Concern Metrics	50
6	COSAAM Transformation Phase	54
6.1	Software Architecture Transformation	54
6.2	Transformation Phase Overview	55
6.3	Activity: Initialize and Sequence Architecture DSM	56
6.3.1	DSM Representation of Module Relationships	56
6.3.2	Window Manager Architecture DSM	59
6.4	Activity: Select Transformation Rule	61
6.4.1	Primitive DMM and DSM Transformations	61
6.4.2	Transformation Rules	62
6.4.3	Transformations rules and primitives	69
6.4.4	Heuristics for applying transformation rules	69
6.4.5	Transforming the Window Manager Software Architecture	70
6.5	Activity: Apply Transformation Rule	71
6.6	Stopping Criteria	74
7	Evolution of the Window Manager Software Architecture	75
7.1	Localizing the Process Management concern	75
7.2	Generalizing Process Termination and Process Management . . .	76
7.2.1	Analysis & Transformation	76
7.3	Decomposing the Window Manager	77
7.4	Defining the Event Management Concern	80
7.4.1	Decomposing the Event Manager	82
7.5	Designing an Operating System Bridge Aspect	84
7.6	Designing a Failure Management Aspect	86
7.7	Designing a new Monitoring aspect	88
8	Discussion and conclusions	92
8.1	Summary	92
8.2	Discussion	92
8.3	Conclusions	94
8.4	Future Work	95

Chapter 1

Introduction

This chapter provides an overview of this thesis. Section 1.1 provides a short introduction to software architecture evaluation. Section 1.2 describes the problem statement of this thesis. Section 1.3 provides an introduction to Design Structure Matrix (DSM) and the Domain Mapping Matrix (DMM), which are tools that are used in our approach. Section 1.4 describes the main contributions of this thesis.

1.1 Software Architecture Evaluation

Software architectures are used to manage the inherent risks in the development of complex software systems. A software architecture communicates early design decisions among stakeholders, and it is used to derive a schedule for software development. Software architectures can be evaluated with software architecture evaluation methods to predict the quality of a design before it is implemented. By evaluating the quality of software architecture designs, stakeholders can further reduce risks and identify trade-offs in the current design.

1.2 Problem Statement

In the past decades, several software architecture evaluation methods have been developed. We identify several problems with existing software architecture evaluation methods. In general, software architecture evaluation methods use scenarios defined by stakeholders to evaluate the quality of software architectures. However, it is generally agreed that the explicit identification and modeling of concerns contribute the design of software architectures. Furthermore, the identification of concerns helps to identify crosscutting concerns. To cope with crosscutting at the architectural level, concerns should be treated as first-class entities, both during design and evaluation. Existing software architecture evaluation methods provide implicit support for

transformation and refactoring of software architecture designs. As a result, it becomes difficult to control the evolution of software architecture designs.

1.3 The Design Structure Matrix and Domain Mapping Matrix

The Design Structure Matrix (DSM) and Domain Mapping Matrix (DMM) are general purpose tools for analyzing complex systems, such as processes, projects, product architectures and software architectures. These tools enable explicit reasoning about the dependencies in and among the elements of complex systems. DSMs show the dependencies among elements in a system, while DMMs show mappings between elements of different domains or systems. Complex systems such as processes can be optimized by *Sequencing* activities in a DSM representation. DSMs have been invented by Steward [38] DSMs have been applied in many domains for all kinds of purposes. The main applications of DSMs are project management, product architecture design and organizational design.

1.4 Contributions

This thesis provides the following contributions to the ongoing research on software architecture evaluation:

1. *Method for evaluating and transforming software architectures*

COSAAM is an iterative method for evaluating and transforming software architecture designs based on DSM and DMM. COSAAM provides a simple concern identification process based on clustering of similar scenarios. Additionally, several heuristics are provided to transform software architectures.

2. *Concern identification process based on clustering of scenarios*

It is generally agreed that a knowledge of stakeholders' concerns is beneficial to defining software architecture designs. COSAAM provides an algorithm for identifying concerns based on DSM clusters of similar scenarios.

3. *Characterization of concerns and modules mapping and measurement of scattering and tangling*

COSAAM provides a characterization of the mappings of concerns and modules and measures of scattering and tangling, based on a DMM. Several method rules are defined to perform this characterization.

4. *COSAAM shows evolution of software architectures*

The DSM and DMM approaches used by COSAAM provide useful ways to monitor the evolution of software architectures.

1.5 Outline of this thesis

In chapter 2 we elaborate on these problems in the context of a comparison between existing software architecture evaluation methods. Additionally, we illustrate these problems by the use of a case study of a software architecture of a window manager system. Based on this analysis we determine requirements for an improvement software architecture evaluation method.

Chapter 3 provides background information on DSMs and DMMs, which are general purpose analysis tools that have been applied in software architecture design. We explain the use of these tools and provide an overview of related work on the application of DSMs to software architecture design. During the last few years, this field has been actively researched.

Chapter 4 introduces the Concern-Oriented Software Architecture Analysis Method. This is a iterative method for evaluating and transforming software architectures. The method consists of the three phases: *preparation*, *analysis* and *transformation*. This chapter introduces the preparation phase and applies its activities to the window manager software architecture.

Chapter 5 introduces the *analysis* phase that is uses a DMM of mappings between concerns and modules. The analysis performed in this phase consists of a characterization of the mappings between concerns and modules and measurements of scattering and tangling of concerns and modules. The activities of the *analysis* phase are applied to the window manager software architecture.

Chapter 6 describes the *transformation* phase that provides several transformation rules to systematically transform the software architecture, and reduce scattering and tangling during the process. A DSM representation of the software architecture is used for dependency analysis of modules. Additionally, several heuristics are defined to help decide between the application of alternative transformation rules. This chapter applies the activities of the *transformation* phase to the window manager software architecture.

Chapter 7 describes eight additional iterations of COSAAM, which consists of the activities of the *analysis* and *transformation* phase. In every iteration we show how the DMM and DSM are updated. At the end of this chapter, we show the final software architecture design of the window manager software architecture design.

Chapter 8 concludes this thesis with a discussion of the development and application of COSAAM. In addition it describes interesting observations about the evolution process of the window manager software architecture.

Chapter 2

Problem Statement

This chapter describes the problem statement of this thesis. Section 2.1 provides an introduction to software architecture evaluation methods and introduces SAAM, one of the earliest methods [21]. Section 2.2 discusses the principle of separation of concerns and crosscutting concerns. In addition, it explains ASAAM, a software architecture evaluation method for identifying aspects at the architecture level. Section 2.3 describes the window manager case study that is used to demonstrate ASAAM and is also used later in the thesis. Section 2.4 discusses several problems of existing software architecture evaluation methods, based on the ASAAM evaluation, comparisons of methods in literature, an evaluation of early aspect approaches and lessons learned during the development of ASAAM-T [4], the tool environment of ASAAM. Section 2.5 defines the requirements for a new software architecture evaluation method based on the analysis of the problems that were identified.

2.1 Software Architecture Evaluation Methods

2.1.1 Introduction

The development of a complex software system carries a large initial investment and considerable risk. This situation increases the need for the development of software architectures. Software architectures are important artifacts that embody the early design decisions which impact the later phases of software development. These early decisions can reduce risk and guide the software development process.

The importance of software architecture designs lead to the need for ways to predict the quality of a design before it is implemented. These kind of evaluations can be performed by using a software architecture evaluation method. In the past decades, several software architecture evaluation methods have been developed. Kazman and others have developed SAAM and ATAM [21] [33]. SAAM is a widely applied scenario-based evaluation methods. SAAM

evaluates the modifiability of software architectures. ATAM is a similar method that is used to identify trade-offs between quality factors in architectural designs. These methods have evolved into more elaborate methods, such as SAAMCS and SAAMER which evaluate complex scenarios, or use evaluate software architectures with respect to different quality attributes, such as evolution and reuse, respectively [5].

2.1.2 SAAM

SAAM is one of the earliest software architecture evaluation methods and is the foundation of many existing scenario-based methods that are used today. Many of its activities are in some way represented in other methods. Therefore, this section introduces the activities of SAAM to get an understand of scenario-based software architecture evaluation methods. Figure 2.1 provides a diagram with the activities of SAAM. There are four activities. The two preliminary activities are *Scenario Development* and *Architecture Description* in which the main artifacts of the methods are derived. The scenarios are then evaluated with respect to the software architecture in the activity *Individual Scenario Evaluation*. Finally, scenario interactions are analyzed in the *Assess Scenario Interactions* activity. SAAM can be used to evaluate a single architecture or to compare different alternative designs.

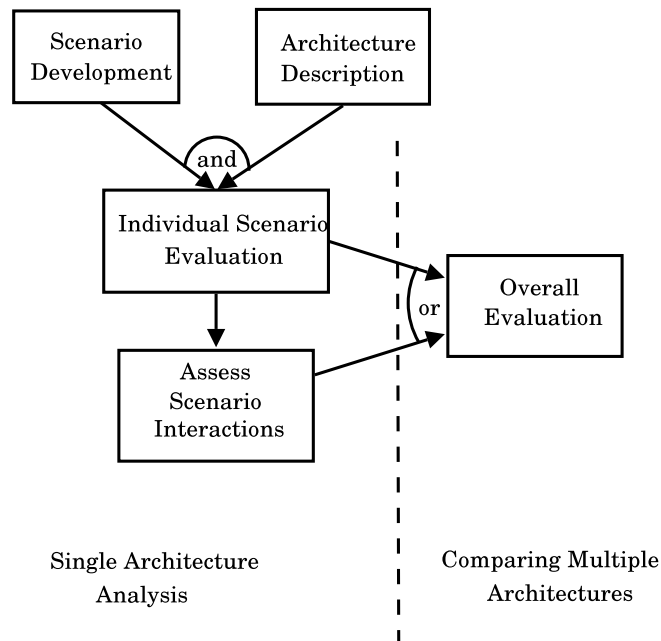


Figure 2.1: Activities of SAAM

2.2 Separation of Concerns and Crosscutting Concerns

Separation of concerns is a fundamental principle in software engineering and other problem-solving activities [32]. This principle states that each concern or problem should be addressed separately by designing a corresponding solution that is relatively independent from solutions for other concerns. The resulting software system designed as a composition of these solutions. By addressing each concern separately, the complexity of the design is reduced and it becomes adaptable.

Crosscutting concerns, such as logging, persistence and error handling, can complicate the separation of concerns in software systems. Crosscutting concerns are difficult to separate from other concerns and may introduce scattered code fragments, which are problematic, as they complicate explicit reasoning about individual concerns [29]. As a result, crosscutting may have a negative impact on the modifiability and maintenance of software systems [44].

Aspect-oriented approaches, such as AspectJ and Composition Filters have been introduced to define modular representations for such concerns. Aspect-Oriented Software Development (AOSD) is a broad field of research on crosscutting concerns and involves methods, tools, languages and other means to deal with crosscutting at the implementation, detailed design and architecture levels [16] [1]. The identification of crosscutting has been incorporated in software architecture evaluation with the development of ASAAM [43]. This is a method for identifying aspects at the architecture level, which is discussed in the next section.

2.2.1 Aspectual Software Architecture Analysis Method

The Aspectual Software Architecture Analysis Method (ASAAM) is a software architecture evaluation method for identifying aspects at the architecture level, [43]. Figure 2.2 shows the activities for ASAAM, which are described briefly.

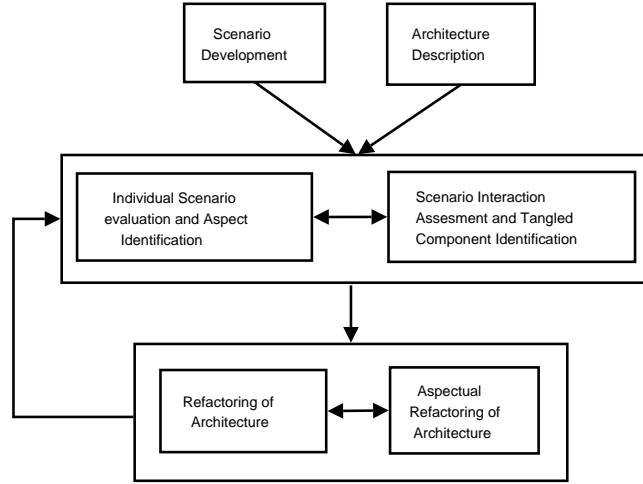


Figure 2.2: Activities of ASAAM

1. *Architecture Description*
A candidate software architecture design is provided that will be evaluated.
2. *Scenario Development*
Scenarios are developed by various stakeholders.
3. *Individual scenario evaluation and aspect identification*
Scenarios are categorized into direct or indirect scenarios. Additionally, aspectual scenarios are identified that indicate possible crosscutting concerns.
4. *Scenario interaction assessment and component classification*
The goal of this step is to determine whether the architecture has an appropriate separation of concerns. Components are classified as *Cohesive*, *Composite*, *Tangled* or *Ill-defined* components.
5. *Refactoring of the architecture*
Based on the previous activity, several refactorings are proposed. These can be conventional refactorings or aspectual refactorings.

The following section introduces the window manager cases study from that is used to demonstrate ASAAM.

2.3 Window Manager Case Study

This section describes the window manager software architecture from the original ASAAM publication [4]. Figure 2.3 shows a UML diagram of the window manager software architecture design.

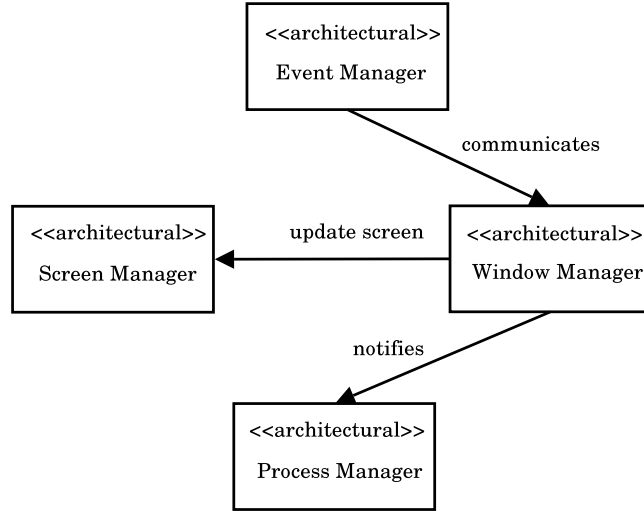


Figure 2.3: Software architecture design of a window management system

The window manager software architecture consists of four modules, the *Event Manager*, *Window Manager*, *Screen Manager* and *Process Manager*. The software architecture works in the following manner: when a user interacts with the system by, for example moving an opened window, system events are generated. The *Event Manager* captures these events and communicates with the *Window Manager* to determine required changes. The *Window Manager* is responsible for performing changes related to windows. Meanwhile, it notifies the *Process Manager* that is responsible for starting, stopping and interrupting processes associated with the windows. When all necessary changes have been performed, the *Window Manager* commands the *Screen Manager* to update the screen.

2.3.1 ASAAM Evaluation of the Window Manager Software Architecture

The following subsections apply all activities of ASAAM to the window manager software architecture.

Activity: Develop Scenarios

In this subsection the scenarios for the window manager software architecture are described. Scenarios can be developed along with other requirements through group activities such as requirements sessions. We use the existing selection of scenarios for the window manager software architecture from the ASAAM publication [43]. Figure 2.4 shows the collection of direct and indirect

scenarios for the window manager software architecture. *Direct* scenarios can be performed by the current software architecture design, while *indirect* scenarios require changes to the design in order to be performed.

Direct Scenarios

- S1. Start multiple processes at the same time.
- S2. Change color of widgets in a window.
- S3. Close all open windows.
- S4. Change screen resolution.
- S5. Enter a command to start an application process.
- S6. Move the main window.
- S7. Screen saver is activated.
- S8. Resize a window.
- S9. Terminate a process.
- S10. Interrupt a process.

Indirect Scenarios

- S11. Change look-and-feel style on run-time.
- S12. Add voice control.
- S13. A failure occurs and the system shuts down.
- S14. Provide dual display screen.
- S15. Use multiple desktops.
- S16. Monitor activities of the user.
- S17. Provide touch screen and light pen as input.
- S18. A memory overflow due to many opened windows.
- S19. Port system to command-based operating system.
- S20. Minimize windows after idle time.

Figure 2.4: Scenarios for the window manager software architecture

Activity: Individual scenario evaluation and aspect identification

Figure 2.5 shows the individual evaluation of scenarios with respect to the modules of the window manager software architecture.

Module	Direct Scenarios	Indirect Scenarios
Event Manager	S3,S5	S12,S13,S16,S17,S18,S19
Screen Manager	S4,S7	S16,S18,S19,S20
Window Manager	S2,S3,S6,S8	S11,S14,S15,S16,S19
Process Manager	S1,S3,S9,S10	S13,S16,S19

Figure 2.5: Scenario interactions for the window manager software architecture

ASAAM provides method rules for identifying aspectual scenarios, based on scattering of scenarios across multiple components. Figure 2.6 shows aspectual scenarios of the window manager software architecture.

- S13. A failure occurs and the system shuts down
- S16. Monitor activities of the user
- S19. Port system to command-based operating system

Figure 2.6: Aspectual scenarios for the window manager software architecture

Activity: Component classification

Figure 2.7 shows the classification of the modules of the window manager software architecture. The classification is based on the ASAAM method rules for component classification and distinguishes components in the categories *Cohesive*, *Tangled*, *Composite* and *Ill-defined*.

Module	Cohesive	Tangled	Composite	Ill-defined
Event Manager	S5	S13,S16,S19	S12,S17	-
Screen Manager	S14	S16,S19	S4,S7	-
Window Manager	S2,S3,S6,S8	S16,S19	S11,S18,S15	-
Process Manager	S1,S9,S10	S16,S19	-	-

Figure 2.7: Classification of the modules of the window manager software architecture

2.4 Problems with existing software architecture evaluation methods

In the previous section we have provided an example of an ASAAM evaluation. In this section we define several problems of existing software architecture evaluation methods based on a comparison of existing methods [5], combined with lessons learned from the development of the ASAAM-T tool environment [4].

1. Concerns are implicitly defined

Scenario-based evaluation methods use scenarios to evaluate software architectures. In general, these methods do not describe stakeholders' concerns explicitly. Concerns are implicitly defined in scenario descriptions. The explicit description of concerns can help to reduce complexity, increase the goal-directness of the development process and provide explicit reasoning [3]. Knowledge of concerns is required to be able to refactor and modify software architectures and we want to answer questions such as: Which concerns are addressed by each individual module?, Which concerns are not yet solved?, Are there any crosscutting concerns? These

questions can be answered when we have identified the stakeholders' concerns for the software architecture.

2. Reusable knowledge is not sufficiently captured

A software architecture evaluation is an effective way to share knowledge about the concerns of stakeholders. However, in the above mentioned evaluation methods, the knowledge gained during the evaluation is not sufficiently captured and modeled for reuse [5]. It is generally agreed that the use of reusable knowledge can reduce development costs and reduce faults in software implementations.

3. Software architecture refactoring and modification is described implicitly

In ASAAM, the process of refactoring and modification of software architectures is described implicitly. Software architecture refactoring is important to improve the separation of concerns and achieve high cohesion in and low coupling between modules. It is generally agreed that these qualities reduce complexity and eases the adaptability of resulting software implementation. However, the refactoring and modification is a complex subject and requires a systematic approach.

4. Limited traceability of concerns

In general, scenario-based software architecture analysis methods like SAAM and ASAAM do not use concerns as first-class entities. As a result, it becomes difficult to establish traceability of concerns through different phases in the software development life-cycle. Traceability of concerns across the software life-cycle, among other factors, improves the ease of evolution of software systems [?]. Traceability of concerns is required in order to refactor and modify software systems. When we can trace earlier design decisions, we may determine how to improve the current design. The problem of limited traceability was identified in an evaluation of early aspect approaches [20]. Recently there has been considerable interest in research on traceability of concerns across the software development life-cycle [6] [22] [44]. We can investigate current research findings to learn how to establish traceability in software architecture evaluation methods.

2.5 Requirements for a software architecture evaluation and transformation method

To cope with the above stated problems a new software architecture evaluation and refactoring method needs to be designed that has the following requirements:

- **The method is a concern-oriented process**

Concern-oriented processes are problem solving processes that use concerns as first-class entities [3]. Concern-oriented processes have several benefits

over ordinary processes such as goal-directness and explicit reasoning [3]. In existing scenario-based analysis methods, explicit reasoning about the evaluation is more difficult due to the absence of concerns as first-class entities.

- **The method supports identification of crosscutting concerns**

ASAAM identifies architectural aspects with aspectual scenarios. Our method should define a process for identification of crosscutting concerns at the architecture level. This requires to make accurate refactoring decisions and decide when to apply aspect-oriented approaches.

- **The method supports conventional and aspect-oriented refactoring**

The method should define a software architecture refactoring process. In this process, the evaluation team can choose between several refactoring strategies. Additionally, heuristics should be defined for selecting an appropriate refactoring strategy for the given situation.

- **The method supports traceability of concerns**

The stakeholders' concerns should be traceable to the software architecture design. This is an example of forward traceability. Additionally we would like to have backwards traceability as well. With backwards traceability we can identify which concerns are supported by modules in the software architecture [6]

Chapter 3

Design Structure Matrices and Domain Mapping Matrices

This chapter introduces the general purpose analysis tools Design Structure Matrix and Domain Mapping Matrix. In section 3.1 we provide a general introduction to the Design Structure Matrix (DSM). Section 3.2 discusses the Domain Mapping Matrix (DMM), a tool that complements a DSM. Section 3.3 discusses related work on the application of DSMs to software architecture design. Section 3.4 discusses recent work on the DSMs in the context of Aspect-Oriented Software Development.

3.1 Design Structure Matrices

A DSM is a matrix representation of the dependencies between elements of a complex system, introduced by Steward [38] in 1981. DSMs are used to represent and analyze complex systems. Figure 3.1 shows an example of a simple DSM of a fictional system. The rows and columns represent subsystems and the **X** marks represent binary dependencies between subsystems. Reading across the columns we can see which subsystem depends on other subsystems. For example, we can see that subsystem A depends on subsystem B and vice versa and that subsystem B depends on subsystem C. Reading across the rows we see which subsystems provide services for other subsystems. We see that subsystem D is not used by any subsystem.

		1	2	3	4
A	1		X		
B	2	X			
C	3		X		X
D	4				

Figure 3.1: Example DSM showing subsystem dependencies

The compact representation of a DSM is useful in and of itself to represent complex systems. The main benefit of DSMs, however, is that they can be manipulated to optimize the system it represents. For example, if the system is a process of interdependent activities, the DSM can be *sequenced* to determine a more efficient sequencing of activities. Sequencing is used to optimize product development schedules and reduce cycle-time [10], [46] Various sequencing algorithms have been defined in literature to sequence DSMs.

Another application of DSMs is *clustering*. The rows and columns are rearranged in a way that clusters of tightly coupled elements are shown. Clustering is used to reduce the complexity of a system. Clustering is applied in the development of product architectures from interdependent components [10] [36]. DSMs have been applied in many different application domains, such as project management [37], product design [36] and more recently, software architecture design [40], [27] and [39].

DSMs are classified as either *static* or *dynamic*. Static DSMs show structural dependencies between elements of a complex system, while dynamic DSMs indicate time-flows between elements. Static DSMs are *component-based* or *team-based* show dependencies between components and teams [10]. Examples of dynamic DSMs are *task-based* and *parameter-based* DSMs, that are used to depict the ordering of activities and design decisions, respectively.

3.2 Domain Mapping Matrices

A DMM is a matrix representation of a mapping between elements from different domains. DMMs complement DSMs, which show dependencies of a system in a single domain, while DMMs show the mapping of elements of two domains. Danilovic introduced the concept of DMM and investigated its application to management of multiple project situations with shared resources [14]. Figure 3.2 shows an example DMM with the mappings of the domains A and B. Each domain has its own elements. Domain A consists of elements A1 up to and including A5. Domain B has four elements, B1 up to and including B4. The cells in the DMM denote mappings between elements of these domains. The semantics of these mappings depends on the context and the meaning of the domains.

		<i>B</i>			
		B1	B2	B3	B4
<i>A</i>	A1	•			
	A2		•	•	
	A3		•		•
	A4	•		•	•
	A5		•		

Figure 3.2: Example Domain Mapping Matrix showing mappings from elements of domain A to domain B

In [22], a traceability matrix is used to identify crosscutting between elements of different phases of the software life-cycle. A traceability matrix can be seen as an instance of a DMM. As far as we know, there are little documented examples of applications of DMMs in literature. Clustering of DMMs has been described, but not made explicit [14]. Further research on DMMs may lead to better understanding on analysis techniques, the same way as was done in research on DSMs.

3.3 DSMs and Software Architecture Design

Recently, DSMs have been used to analyze and manage software architectures [35] [40]. Lattix inc. has developed a tool called LDM, which can manage a DSM derived from structural dependencies in a Java or C/C++ implementation [24]. Figure 3.3 shows an example of LDM with a DSM of the dependencies of the packages of Apache ant.

\$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
org.apache.tools	+ cvslib	1																
	+ ant.taskdefs	2					2											
	+ rmic	3					2											
	+ condition	4					12				2	3	1					
	+ email	5					1											
	+ *	6	5	7	4	3												
	+ listener	7																
	+ ant	8												1				
	+ helper	9					3							4				
	+ input	10					3					12	1					
	+ filters	11	4	19	7		3	152			17		2	9				
	+ types	12	1	3	3	1		55	1	1	4	13		12				
	+ util	13	11	25	14	20	10	309	4	12	3	6	71	13				
util	+ org.apache.tools.bzip2	14					4											
	+ org.apache.tools.mail	15				1		1										
	+ org.apache.tools.tar	16					4											
	+ org.apache.tools.zip	17					5											

Figure 3.3: Lattix LDM tool with a DSM of the dependencies between the packages of *Apache Ant*

In the past few years, there has also been a growing interest in using DSMs and Net Option Value to determine the value of competing software architecture designs. Net Option Value is a economic model of real options that is used to analyze the modularity of design structures. Lopes and Bajracharya and Sullivan et. al. have focused on determining the value of conventional software architecture designs [27] [40].

3.4 DSMs and Aspect-Oriented Software Architecture Development

The application of Design Structure Matrices to aspect-oriented software architecture design is relatively new research area and is discussed in [27] and [39]. Lopes and Bajracharya also demonstrate that aspects can make software architecture designs more valuable. However, in [26] they show that aspects can however provide a negative contribution to the value of a software architecture design. These conclusions are supported by applying the Net Option Value and DSM techniques to a set of conventional and aspect oriented software architecture designs. In this work, the authors also introduce preliminary design guidelines for aspects.

Chapter 4

The Concern-Oriented Software Architecture Analysis Method

In this chapter we provide an overview of the Concern-Oriented Software Architecture Analysis Method (COSAAM). Section 4.1 describes the preparation, analysis and transformation phases of COSAAM. Section 4.2 provides a detailed description of the activities that are performed in the preparation phase. In the sections 4.2.1 and 4.2.2 we explain and perform the activities of the preparation phase to prepare the window manager software architecture for a COSAAM evaluation.

4.1 Overview of COSAAM

COSAAM is an iterative software architecture evaluation and transformation method. A software architecture evaluation is usually performed by an *evaluation team*, which consists of stakeholders, software architects and possibly other people involved in a software development project [12]. Figure 4.1 shows the UML activity diagram [18] of the phases of COSAAM. COSAAM consists of the three phases: the *Preparation Phase*, *Analysis Phase* and *Transformation Phase*.

The preparation phase establishes the artifacts used in the COSAAM evaluation: a candidate software architecture design and a collection of concerns of stakeholders. The analysis phase involves a characterization and measurement of scattering and tangling of concerns and modules. The information provided during this analysis is used in the transformation phase, in which the candidate software architecture is transformed. An iteration of COSAAM consists of all the activities of the analysis and transformation phases. The evaluation

team can decide to perform a new iteration, or stop the evaluation process. We describe several stopping criteria in the transformation phase to guide this decision making.

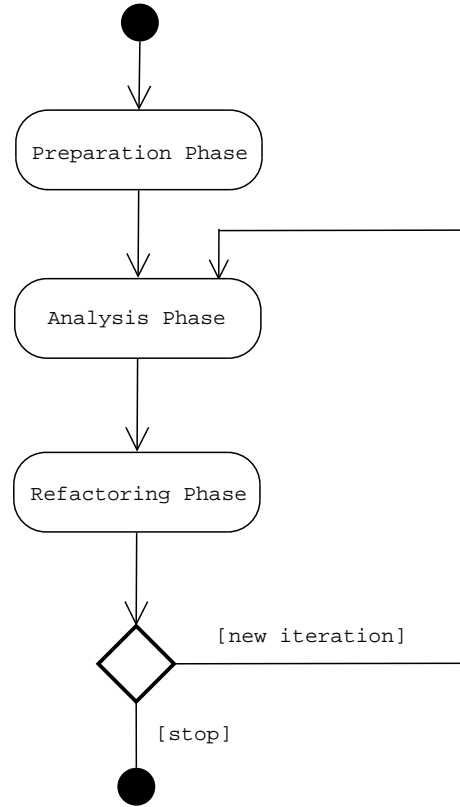


Figure 4.1: UML activity diagram of the phases of COSAAM

In the following paragraph we explain the *preparation*, *analysis* and *transformation* phases of COSAAM based on their goals, inputs and outputs.

1. *Preparation Phase*

- *Goal*

The goal of the preparation phase is to define the concerns of stakeholders and deliver a candidate software architecture design. The candidate software architecture will be evaluated in the analysis and transformation phases.

- *Inputs*

The inputs for this phase are existing concerns or a reused software architecture design.

- *Outputs*

The outputs for this phase are a collection of concerns of stakeholders and a candidate software architecture design.

2. *Analysis Phase*

- *Goal*

The goal of the analysis phase is to determine how the candidate software architecture addresses the concerns of stakeholders. Additionally, scattering and tangling of concerns and modules is measured. The results of this analysis is used for decision making in the consecutive transformation phase.

- *Inputs*

The inputs of the analysis phase are a collection of concerns and a candidate software architecture design.

- *Outputs*

The analysis phase delivers a concern-module DMM representation of the mapping of concerns to the modules of the software architecture. The DMM is enhanced with a characterization of concerns and modules and a collection of metrics that measure scattering and tangling of concerns and modules in the candidate software architecture.

3. *Transformation Phase*

- *Goal*

The goal of the transformation phase is to address the concerns of stakeholders by transforming the candidate software architecture. At the same time, scattering and tangling of concerns and modules is reduced.

- *Inputs*

The inputs of the transformation phase are the candidate software architecture design and the concern-module DMM enhanced with a characterization of concerns and modules and the measurements of scattering and tangling of concerns and modules.

- *Outputs*

The outputs of the transformation phase are a DSM representation of the candidate software architecture and an updated DMM with the mapping of concerns to modules.

4.2 Preparation Phase Overview

The preparation phase aims to derive concerns and deliver a candidate software architecture design. Figure 4.2 shows the five activities of the preparation phase. In the following subsections we will explain all the goals, inputs and outputs of these activities. We apply each activity to the window manager case study from the ASAAM publication [43]. Subsection 4.2.1 describes the *Define Concerns* activity that consists of the sub-activities *Reuse Concerns*, *Develop Scenarios*, *Initialize and Cluster Scenario DSM* and *Derive Concerns*. Finally, section 4.2.2 describes the *Describe Candidate Software Architecture* activity, where we deliver as candidate software architecture. In this case it is the window manager software architecture from chapter 2.

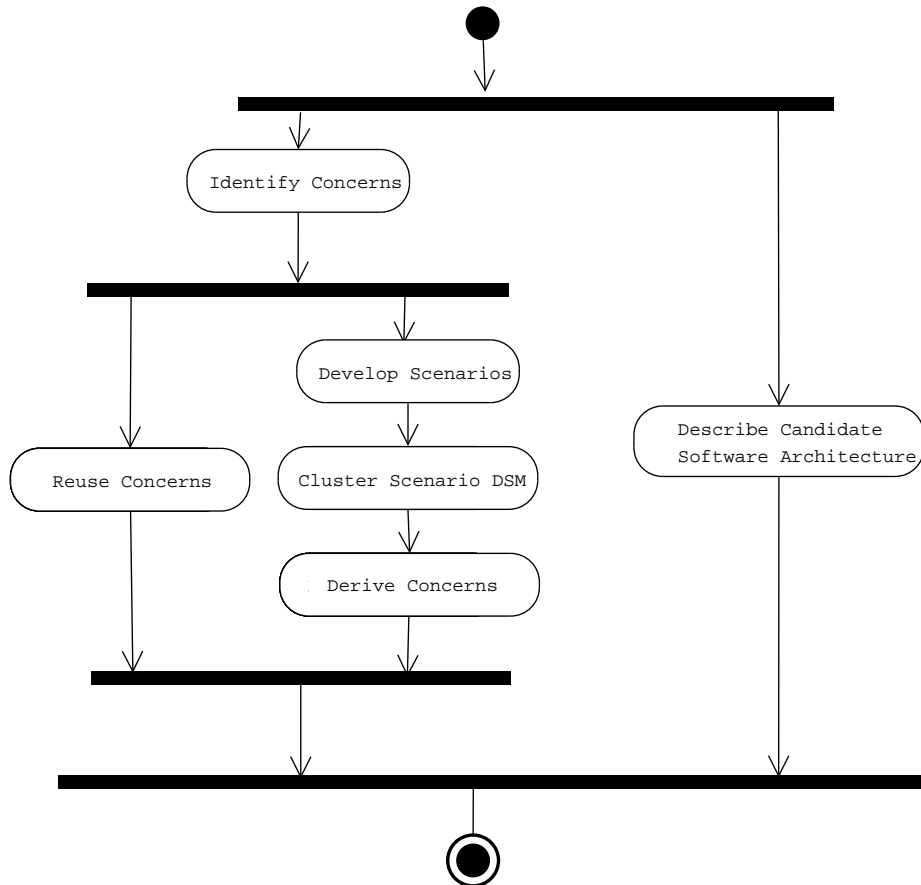


Figure 4.2: UML activity diagram of the preparation phase of COSAAM

4.2.1 Activity: Define Concerns

- *Goal*

The goal of this activity is to define a collection of concerns of the stakeholders.

- *Inputs*

There are no inputs for this activity.

- *Outputs*

The output of this activity is a collection of stakeholders' concerns.

The goal of this activity is to define a set of concerns that have to be solved by the software architecture. This activity is decomposed into several sub-activities. Concerns can be reused from existing projects, requirements specifications or

domain models in the sub-activity *Reuse Concerns*. They can be also be derived from scenarios developed by stakeholders through the use of similarity analysis and DSM clustering techniques. This process consists of the three sub-activities *Develop Scenarios*, *Initialize and Cluster Scenario DSM* and *Derive Concerns*, that are explained below.

Sub-activity: Reuse Concerns

- *Goal*
The goal of this activity is to reuse concerns.
- *Inputs*
The input of this activity is the documentation of past projects or literature.
- *Outputs*
The outputs of this activity are reused concerns.

Sub-activity: Develop Scenarios

Scenarios are short descriptions of interactions with the software system to be developed [12]. The evaluation team develops or reuses a collection of scenarios that indicate important usage or change situations in the software.

- *Goal*
The goal of this activity is to develop scenarios that describe relevant interactions with the software system.
- *Inputs*
The optional inputs for this activity are reused scenarios.
- *Outputs*
The output of this activity is a collection of scenarios.

The scenarios of the window manager system have been described in chapter 2.

Sub-activity: Initialize and Cluster Scenario Design Structure Matrix

Before we describe the goal, inputs and outputs of this sub-activity we provide a vocabulary of concepts that which refer to:

- *Similarity Relationship*
A similarity relationship is a relationship between two scenarios. Scenarios that are in this relationship share similar features in their descriptions or in semantics.

- *Number of similar scenarios*

The number of similarity relationships a scenario is in.

- *Cluster*

A cluster is a set of scenarios.

- *Overlapping Cluster*

An overlapping cluster is a cluster that includes at least one scenario that is part of another cluster.

- *Concern*

There are many definitions for concerns. We use the definition from Bakker [6], which states: “A *concern* is a concept that refers to a problem that is of interest to one or more stakeholders”. Concerns are derived from the analysis of clusters of scenarios.

To understand the concerns that are important for developing the software architecture, we need a way to identify concerns from the scenarios. We do this by entering scenarios into a DSM. The evaluation team identifies scenarios that share a common domain concept or that describe similar problems. The similarity relationships are documented in the DSM. We then apply clustering to identify clusters of similar scenarios. Based on the analysis of clusters we derive concerns. Our algorithm produces overlapping clusters of scenarios with simple heuristics. A mark in a cell indicates similarity between two scenarios in the corresponding row and column. An empty cell in the DSM indicates that two scenarios are not similar to each other.

- *Goal*

The goal of this activity is to analyze and cluster scenarios in a DSM.

- *Inputs*

The input of this activity is a collection of scenarios.

- *Outputs*

The output of this activity is a collection of clusters of scenarios.

We apply this activity to the scenarios of the window manager software architecture. The evaluation team enters the scenarios in the scenario DSM. Scenarios that seem to address the same concern or share the same domain concepts are linked to each other by marking the appropriate cell in the DSM, shown in figure 4.2.1.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
S1	■				•				•	•										
S2		■									•									
S3			■						•											
S4				■			•			•				•	•					
S5	•				■					•		•								
S6						■		•												
S7				•			■									•				•
S8						•		■				•								
S9	•		•						■	•										
S10	•				•				•	■		•								
S11		•		•							■			•	•					
S12				•			•		•			■					•			
S13													■					•		
S14				•						•				■		•				
S15				•						•					■					
S16							•									■				•
S17												•					■			•
S18													•					■		
S19																			■	
S20							•									•	•			■

Figure 4.3: Scenario DSM for the Window Manager Software Architecture

The DSM depicted in figure 4.2.1 now needs to be clustered in order to derive concerns. We have developed a heuristics-based clustering algorithm to define clusters of scenarios. Existing DSM clustering algorithms were not appropriate to use because most of them are based on dependencies, instead of similarity. Dependency is an asymmetrical relationship, while similarity is a symmetric relationship. When element A is dependent on element B, element B does not necessarily depend on B. However, when element A is similar to element B, element B is also similar to element A. Before we explain the details of our algorithm we present two heuristics for defining clusters.

1. *Similarity Heuristic* Scenarios that are similar to many other scenarios are likely related to the same concern or similar concerns and are put into a cluster with all scenarios in its similarity relationship
2. *Dissimilarity Heuristic* Scenarios that are not similar to any other scenario probably describe a separate concern and are put into a separate cluster

The *similarity heuristic* states that scenarios that have a similar description, for example by using the same domain concepts, describe the same concern. The following example shows that this heuristic can be useful. The scenarios *S1: Start multiple processes at the same time*, *S5: Enter a command to start an application process* and *S10: Interrupt a process* all describe process

management actions: starting, stopping and interrupting processes. The scenarios are descriptions of a group of related process management activities.

The second heuristic is the dissimilarity heuristic, which is the counterpart of the similarity heuristic. This heuristic states that scenarios that describe very different system interactions probably describe a different concern. We provide an example for the window manager case study. Scenario *S19: Port system to command-based operating system* is very different from all other scenarios, as it is the only one that describes porting of the window manager.

These heuristics form the basis of our clustering algorithm. The algorithm proceeds by creating a list of scenarios that is ordered by decreasing degree, i.e. the number of scenarios it is related to. The first elements of the list are elements with the highest degree. The end of the list contains scenarios that are isolated or related to a single other scenario. The algorithm iterates through the list and clusters the scenarios according to the heuristics described above. A scenario is clustered by putting it together with all its neighbors in a cluster. The algorithm performs this procedure for all elements that are not yet in a cluster. Scenarios with a single neighboring scenario are clustered together. Finally, an isolated element is clustered by putting the scenario in its own cluster.

Note that if the evaluation team does not define any similarity relationships, each scenario is put in its own cluster. In that case, each scenario is a separate concern. This algorithm is implemented in ruby [2]. The source code for our algorithm is added as an appendix to this thesis 8.4. We illustrate clustering by providing a graph representation of the scenario DSM in figure 4.4. The graph consists of two large graphs of scenarios, an individual scenario that is not connected to other scenarios, (S19), and a pair of scenarios, (S13 and S18).

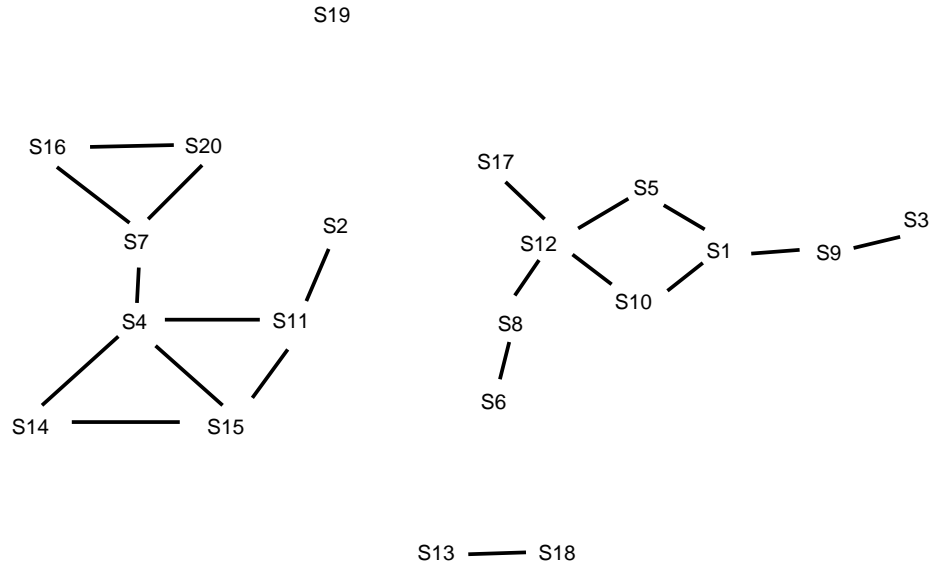


Figure 4.4: Graph representation of the scenario DSM

After applying the algorithm, the DSM is clustered. Figure 4.5 shows the scenario graph with added clusters, delivered by our algorithm.

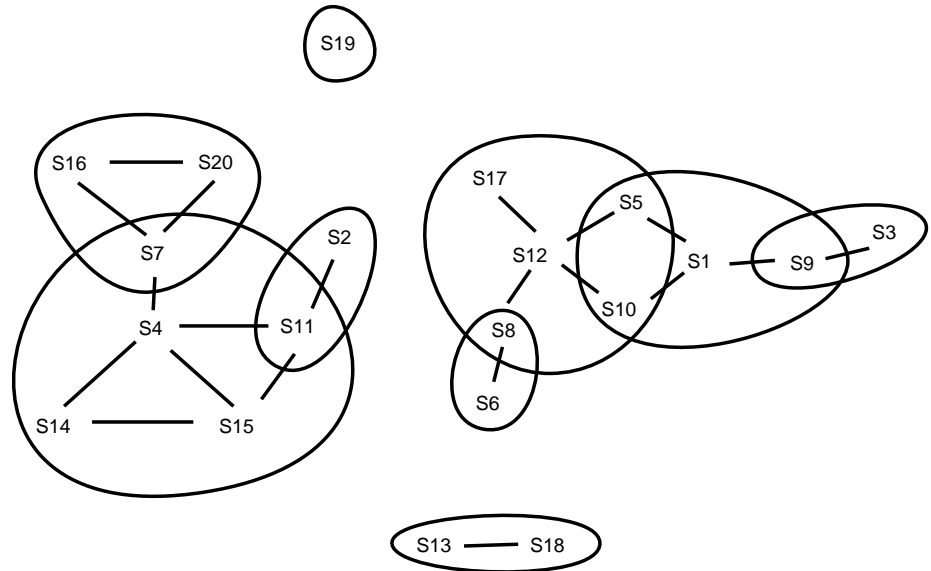


Figure 4.5: Clustered graph representation of the scenario DSM

The graph representation of the clustered DSM can be represented in its original DSM form. The clustered DSM is shown in figure 4.6. The algorithm identified 9 overlapping clusters of scenarios.

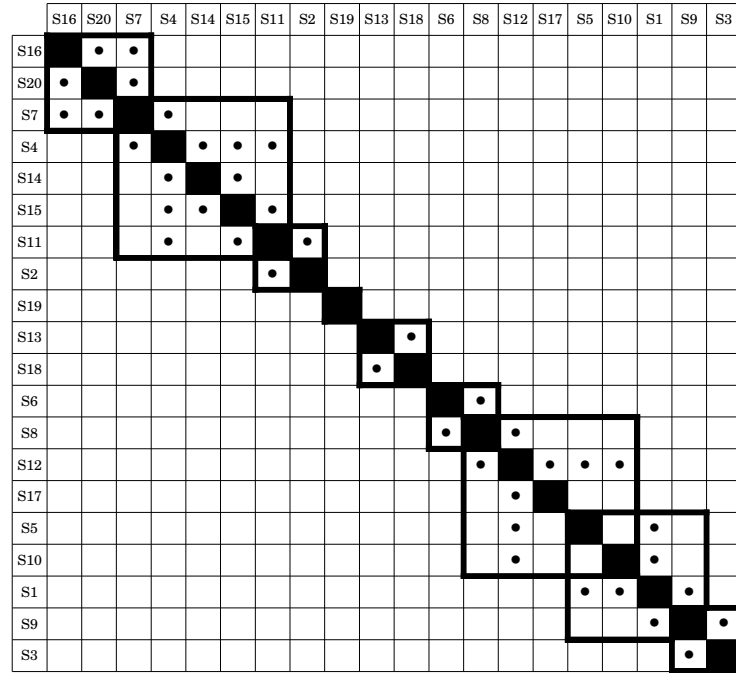


Figure 4.6: Clustered DSM of scenarios for the Window Manager Software Architecture

Sub-activity: Derive Concerns

- *Goal*
The goal of this activity is to derive concerns from clusters of scenarios.
- *Inputs*
The input of this activity is a collection of clusters of scenarios.
- *Outputs*
The outputs of this activity is a collection of concerns.

Our clustering algorithm has delivered 9 clusters of scenarios. In this activity, we identify which concerns these clusters represent. Figure 4.7 shows the graph representation of the clustered scenario DSM with added names of concerns.

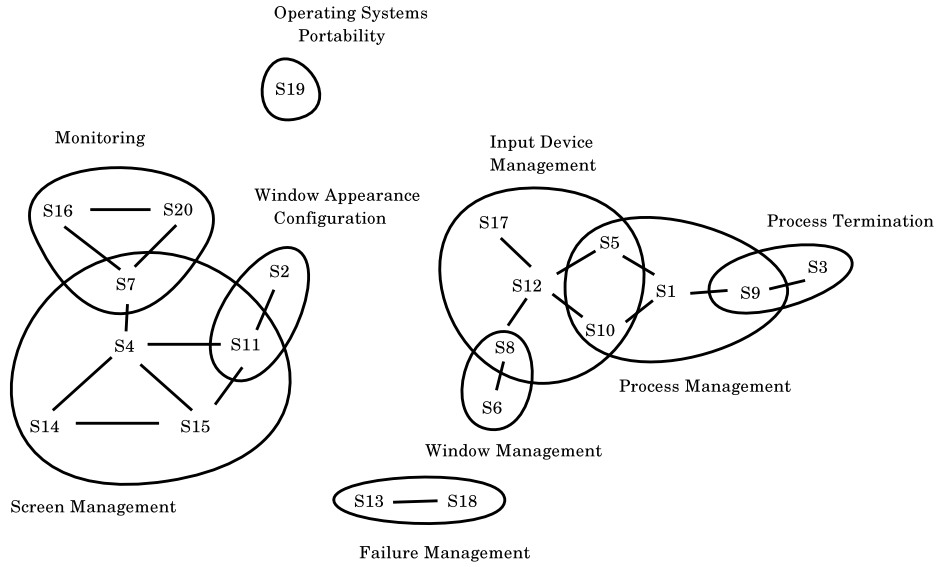


Figure 4.7: Concerns derived from the graph representation of the clustered scenario DSM

Figure 4.8 shows the 9 concerns that we have identified.

<i>Concerns</i>	<i>Scenarios</i>
Monitoring	S16, S20, S7
Screen Management	S7, S4, S14, S15, S11
Window Appearance Configuration	S11, S2
OS Portability	S19
Failure Management	S13, S18
Window Management	S6, S8
Input Device Management	S8, S12, S17, S5, S10
Process Management	S5, S10, S1, S9
Process Termination	S9, S3

Figure 4.8: Derived concerns for the Window Manager Software Architecture

The concerns *Screen Management* and *Process Management* cluster many scenarios. These concerns are well-defined since the scenarios clustered by them clearly indicate specific actions and events that belong to these concerns. In contrast to these concerns, the concern *Process Termination* clusters fewer scenarios. Generally, the more concrete scenarios one can analyze, the easier it is to define a concept.

4.2.2 Activity: Describe Candidate Software Architecture

In this activity, the software architects describe a candidate software architecture. The candidate software architecture will be transformed several times during the evaluation, so it is acceptable if the software architecture is still an initial version. The ambiguity in the design and the definition of concerns will be reduced during several iterations of COSAAM. As a result, the evaluation team will gain more problem and solution domain knowledge, that allows them to improve the candidate software architecture.

- *Goal*

The goal of this activity is to describe a candidate software architecture.

- *Inputs*

The optional inputs of this activity are a reusable software architecture design

- *Outputs*

The output of this activity is a candidate software architecture design.

The candidate software architecture is described in this activity. Software architecture designs usually consist of several views that each show different parts of the total design [11]. COSAAM is in principle agnostic with respect to any kind of architectural view-type. However, in this thesis we use the term *module* when we refer to architectural elements. Figure 4.9 shows the window manager software architecture described earlier in chapter 2.

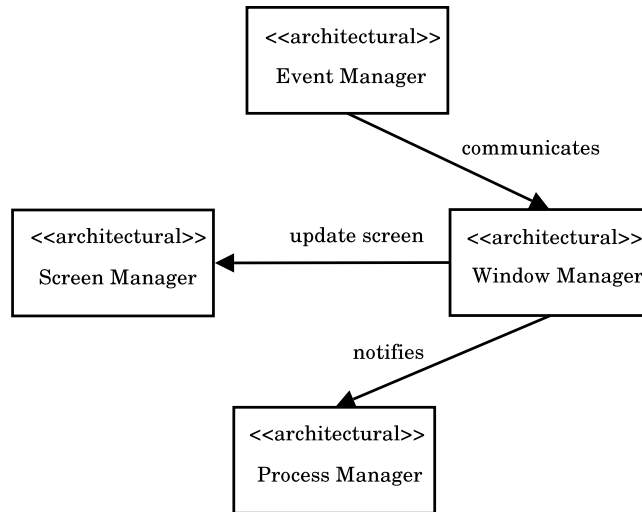


Figure 4.9: Software architecture of a window manager system

Chapter 5

COSAAM Analysis Phase

This section discusses the activities of the COSAAM analysis phase. The goal of the analysis phase is to provide the evaluation team with a characterization of the mapping between concerns and modules and measures of scattering and tangling. This analysis is used in the following transformation phase to make judgments regarding the selection of transformation rules to transform the software architecture. The following section describes the activities of the analysis phase and the details of the analysis that is performed.

5.1 Analysis Phase Overview

Figure 5.1 describes the three activities of the analysis phase. These are the activities *Initialize Concern-Module DMM*, *Characterize Concern and Module Mapping* and *Measure Scattering and Tangling*. Sections 5.1.1, 5.1.2, 5.1.3 explain the goals, inputs and outputs of each activity and provide a detailed description of their application to the window manager software architecture.

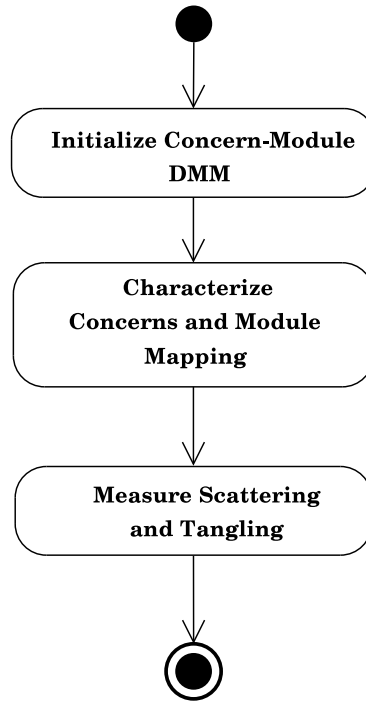


Figure 5.1: UML activity diagram of the activities of the COSAAM analysis phase

5.1.1 Activity: Initialize Concern-Module DMM

- *Goal*

The goal of this activity is to initialize the concern-module DMM. This DMM shows the mapping of concerns to architectural modules. The resulting DMM representation is used in later activities to measure scattering and tangling and guide the activities in the transformation phase.

- *Inputs*

The inputs of this activity are the candidate software architecture and a collection of concerns.

- *Outputs*

The output of this activity is a DMM with the mappings of concerns to modules. The DMM shows which concerns are currently addressed by the software architecture, represented by *direct* mappings, and concerns that require changes in order to be addressed. This situation is represented by *indirect* mappings from concerns to modules.

The evaluation methods SAAM and ASAAM use mappings of scenarios to modules and distinguish between a *direct* scenario and an *indirect* scenario [21], [43]. A *direct* scenario can be performed by the candidate software architecture, while an *indirect* scenario requires changes to existing design. COSAAM uses a similar distinction for the mapping of concerns to modules and introduces a *direct* mapping and an *indirect* mapping between concerns and modules. If a *direct* mapping exists between a concern and a module, it means that the module is responsible for solving this concern. If an *indirect* mapping exists between a concern and a module, this means that this module requires changes in order to solve this concern. When there is no mapping between a concern and a module then that module is not responsible for solving that concern.

To demonstrate this activity the concern-module DMM is created for the window manager software architecture, shown in figure 5.2. The rows represent the stakeholders' concerns. The columns represent the modules of the window manager software architecture. The cells contain either a *direct* mapping, identified by the letter 'D', an *indirect* mapping, identified by the letter 'I', or they do not contain anything. To simplify our analysis some of the concern-to-module mappings have been derived from the existing individual scenario evaluation performed in the ASAAM publication [43]. Note that this is not required, the initialization of the concern-module DMM can be performed without any previous evaluation of scenarios.

		<i>Modules</i>			
		Event Manager (EM)	Window Manager (WM)	Process Manager (PM)	Screen Manager (SM)
<i>Concerns</i>	Monitoring (MO)	I	I	I	I
	Operating Systems Portability (OP)	I	I	I	I
	Failure Management (FM)	I	I	I	I
	Process Termination (PT)	D	D	D	
	Process Management (PM)		D	D	
	Input Device Management (IDM)	I			
	Window Management (WM)		D		
	Window Appearance Configuration (WAC)		D		
	Screen Management (SM)				D

Legend

D	Direct Mapping
I	Indirect Mapping

Figure 5.2: Concern-module DMM for the window manager software architecture

5.1.2 Activity: Characterize Concern and Module Mapping

- *Goal*

The goal of this activity is to characterize the mapping between concerns and modules based on the concern-module DMM.

- *Inputs*

The input of this activity is the concern-module DMM.

- *Outputs*

The output of this activity is a characterization of the concerns and modules based on the mappings in DMM.

In this activity the mapping between concerns and the modules of the candidate software architecture is characterized. Scattering, tangling and crosscutting are identified and whether concerns are directly and indirectly addressed by modules. This characterization enables explicit reasoning about the transformation of concerns and modules in the *transformation* phase. Concern and modules can be characterized by a combination of mappings in the concern-module DMM. There are 7 categories for concerns and 4 categories of modules.

Concern Artifact Diagram

Artifact diagrams are used in method engineering to define the categories of artifacts that are used by a method [41]. Figure 5.3 shows the artifact diagram with the categories and associated method rules used for the characterization of concerns. The boxes printed in bold are the categories used in COSAAM. The other boxes are intermediate categories.

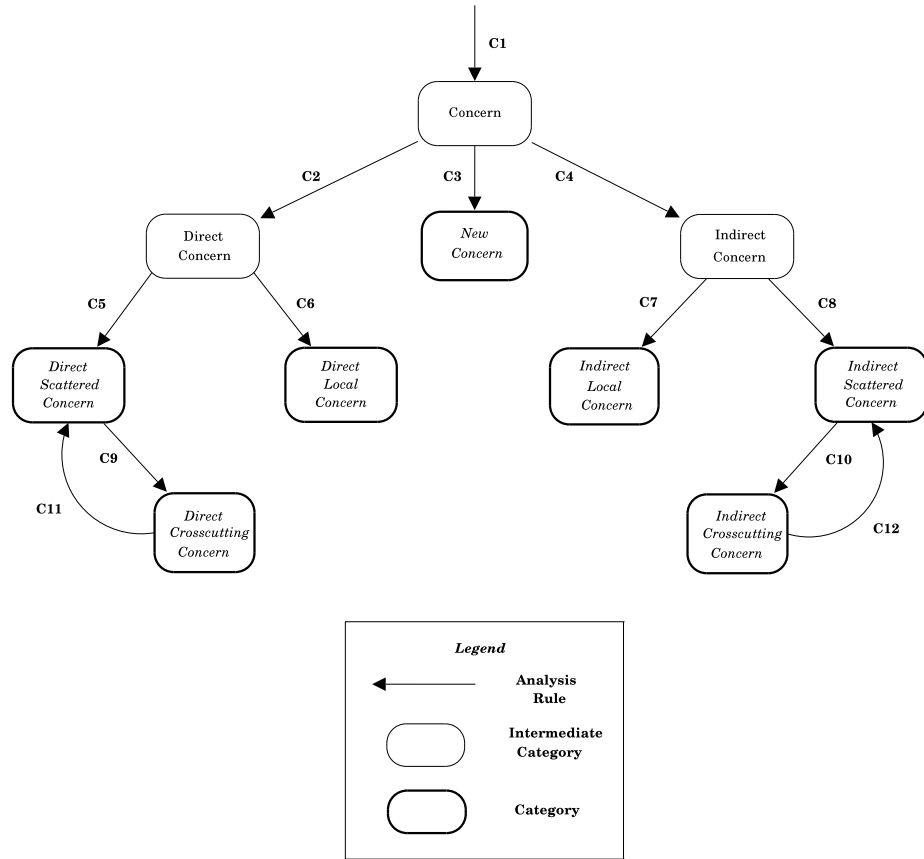


Figure 5.3: Concern artifact diagram

The analysis rules from figure 5.3 are defined in the following format:

IF <condition> THEN <consequent>

Whenever the condition of an analysis rule is met, concern is categorized into a specific category. This representation for method rules was also used for the method rules of ASAAM [43]. The following figure describes the analysis rules of concerns.

C1: SELECT concern from preparation phase

C2: IF concern is directly addressed by a module
THEN concern **is** a DIRECT CONCERN

C3: IF concern is not addressed by any module
THEN concern **is** a NEW CONCERN

C4: IF concern is indirectly addressed by a module
THEN concern **is** an INDIRECT CONCERN

C5: IF DIRECT CONCERN is addressed by multiple modules
THEN concern **is** a DIRECT SCATTERED CONCERN

C6: IF DIRECT CONCERN is addressed by a single module
THEN concern **is** a DIRECT LOCAL CONCERN

C7: IF INDIRECT CONCERN is addressed by a single module
THEN concern **is** an INDIRECT LOCAL CONCERN

C8: IF INDIRECT CONCERN is addressed by multiple modules
THEN concern **is** a INDIRECT SCATTERED CONCERN

C9: IF DIRECT SCATTERED CONCERN is addressed by at least one TANGLED MODULE
THEN concern **is** a DIRECT CROSSCUTTING CONCERN

C10: IF INDIRECT SCATTERED CONCERN is addressed by at least one TANGLED MODULE
THEN concern **is** an INDIRECT CROSSCUTTING CONCERN

C11: IF DIRECT CROSSCUTTING CONCERN is not an inherent crosscutting concern
THEN concern **is** a DIRECT SCATTERED CONCERN

C12: IF INDIRECT CROSSCUTTING CONCERN is not an inherent crosscutting concern
THEN concern **is** an INDIRECT SCATTERED CONCERN

Figure 5.4: Concern analysis rules

- *New Concern (N)*
A *New Concern* is not mapped to any module in the candidate software architecture. Figure 5.5 shows a DMM example of a new concern.

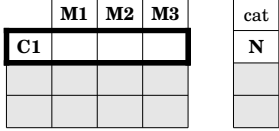
<i>DMM Mapping</i>	<i>Category</i>
	New Concern (N)

Figure 5.5: DMM mapping of a *New Concern* (N)

- *Direct Local Concern (DL)*

A *Direct Local Concern* is directly mapped to exactly one module. Figure 5.6 shows a DMM example of the direct local concern.

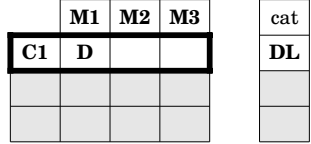
<i>DMM Mapping</i>	<i>Category</i>
	Direct Local Concern (DL)

Figure 5.6: DMM mapping of a *Direct Local Concern* (DL)

- *Indirect Local Concern (IL)*

An *Indirect Local Concern* is indirectly mapped to exactly one module. Figure 5.7 shows a DMM example of the Indirect Local Concern.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td>M2</td><td>M3</td></tr><tr><td>C1</td><td>I</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td></tr><tr><td>IL</td></tr><tr><td></td></tr><tr><td></td></tr></table>		M1	M2	M3	C1	I											cat	IL			Indirect Local Concern (IL)
	M1	M2	M3																		
C1	I																				
cat																					
IL																					

Figure 5.7: DMM mapping of a *Indirect Local Concern* (IL)

- *Direct Scattered Concern (DS)*

A *Direct Scattered Concern* is directly mapped to multiple modules. Figure 5.8 shows a DMM mapping of the Direct Scattered Concern.

DMM Mapping	Category																				
<table><tr><td></td><td>M1</td><td>M2</td><td>M3</td></tr><tr><td>C1</td><td>D</td><td>D</td><td>D</td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td></tr><tr><td>DS</td></tr><tr><td></td></tr><tr><td></td></tr></table>		M1	M2	M3	C1	D	D	D									cat	DS			Direct Scattered Concern (DS)
	M1	M2	M3																		
C1	D	D	D																		
cat																					
DS																					

Figure 5.8: DMM mapping of a *Direct Scattered Concern* (DS)

- *Indirect Scattered Concern (IS)*

An *Indirect Scattered Concern* is mapped to multiple modules, of which at least one is an *indirect* mapping. Figure 5.9 shows a DMM mapping of the Indirect Scattered Concern.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td>M2</td><td>M3</td></tr><tr><td>C1</td><td>I</td><td>I</td><td>I</td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td></tr><tr><td>IS</td></tr><tr><td></td></tr><tr><td></td></tr></table>		M1	M2	M3	C1	I	I	I									cat	IS			Indirect Scattered Concern (IS)
	M1	M2	M3																		
C1	I	I	I																		
cat																					
IS																					

Figure 5.9: DMM mapping of a *Indirect Scattered Concern (IS)*

- *Direct Crosscutting Concern (DX) or Direct Scattered Concern (DS)*

A *Direct Crosscutting Concern* is scattered over several modules, of which at least one is tangled. If after evaluation the concern does not seem to be *inherently* crosscutting, it becomes a *Direct Scattered Concern*. The same DMM mapping corresponds to these categories. Figure 5.10 shows the DMM mapping of a *Direct Crosscutting Concern* or *Direct Scattered Concern*.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td>M2</td><td>M3</td></tr><tr><td>C1</td><td>D</td><td>D</td><td>D</td></tr><tr><td>C2</td><td>D/I</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td></tr><tr><td>DX, DS</td></tr><tr><td>DL/IL</td></tr><tr><td></td></tr></table>		M1	M2	M3	C1	D	D	D	C2	D/I							cat	DX, DS	DL/IL		Direct Crosscutting Concern (DX) or Direct Scattered Concern (DS)
	M1	M2	M3																		
C1	D	D	D																		
C2	D/I																				
cat																					
DX, DS																					
DL/IL																					

Figure 5.10: DMM mapping of a *Direct Crosscutting Concern (DX)* or *Direct Scattered Concern (DS)*

- *Indirect Crosscutting Concern (IX) or Indirect Scattered Concern (IS)*

An *Indirect Crosscutting Concern* is scattered over several modules, of which at least one is tangled. If after evaluation the concern does not seem to be *inherently* crosscutting, it becomes a *Indirect Scattered Concern*. The same DMM mapping corresponds to these categories. Figure 5.10 shows the DMM mapping of a *Indirect Crosscutting Concern* or *Indirect Scattered Concern*.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td>M2</td><td>M3</td></tr><tr><td>C1</td><td>I</td><td>I</td><td>I</td></tr><tr><td>C2</td><td>D/I</td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td></tr><tr><td>IX, IS</td></tr><tr><td>DL/IL</td></tr><tr><td></td></tr></table>		M1	M2	M3	C1	I	I	I	C2	D/I							cat	IX, IS	DL/IL		<p>Indirect Crosscutting Concern (IX) or Indirect Scattered Concern (IS)</p>
	M1	M2	M3																		
C1	I	I	I																		
C2	D/I																				
cat																					
IX, IS																					
DL/IL																					

Figure 5.11: DMM mapping of a *Indirect Crosscutting Concern* (IX) or *Indirect Scattered Concern* (IS)

Knowledge of crosscutting concerns is important for the transformation phase, since it can influence the ordering of transformations. The formal definition of crosscutting by van den Berg et. al [22] is used to identify crosscutting concerns. The crosscutting definition by van den Berg et al. states: "*Crosscutting occurs, then in a mapping between source and target, a source elements is scattered over target elements and where in at least one of these target elements, on or more source elements are tangled.*" Their crosscutting definition is based on scattering and tangling in traceability matrices. Because our DMM is an instance of a traceability matrix, we can use this definition to identify crosscutting concerns.

The identification of crosscutting based on the above definition may not necessarily indicate crosscutting concerns that need for aspect-oriented composition techniques in order to be solved. We distinguish between *inherent* and *accidental* crosscutting. Inherent crosscutting very likely requires aspect-oriented composition mechanisms. Accidental crosscutting however, is likely the result of problems that can be prevented without using aspect-oriented composition mechanisms. We mention three causes of accidental crosscutting in that can be identified with the DMM: Additionally, we evaluate the crosscutting concerns to identify whether they are *inherently* crosscutting or accidentally crosscutting. This removes any *false positives* from the set of crosscutting concerns. If a concerns is accidentally crosscutting it becomes a *Direct Scattered Concern* or *Indirect Scattered Concern*, if it was a *Direct Crosscutting Concern* or a *Indirect Crosscutting Concern* respectively.

1. Incorrect identification of concerns (Concerns)
2. Faults or constraints in the architectural design (Modules)
3. Faults in the evaluations (Mappings)

The first problem is the incorrect or incomplete definition of concerns in the software architecture. If a definition of a concern remains vague, it is

hard to determine if, and how it is addressed by modules in the software architecture. Nominalization is ‘*the conversion of a set of activities into a single universal noun, such as “measurement”, “efficiency”, or “security”.*’ [45]. Nominalization is unavoidable in software development, since nominalization it is used for abstraction. We use nouns to label concerns, modules, processes, classes and so on. If this nominalization is too ambiguous however, this can give problems. The semantics of words depends on the context. However, due to the iterative nature of COSAAM, the evaluation team learns more about the software architecture and the context and ambiguity in both concerns and modules is reduced.

The second class of problems that can cause accidental crosscutting are faults in the software implementation or software architecture design. Examples of these problems are the well-known collection of anti-patterns in software development [9]. An example anti-pattern is *The Blob*, a module that has too much responsibilities which results in a highly tangled code fragments. However, accidental crosscutting may also be caused by design constraints or carefully made design decisions and trade-offs. Another cause of accidental crosscutting is the use of code generators. Code generators may generate code that provides useful features such as persistence or notification. The code fragments however may be scattered across several classes which seems to be the result of crosscutting, but are not necessarily.

Finally, the third possible cause of accidental crosscutting is the existence of faults in the evaluation. This problem can occur when there is little architectural design information. This can happen during the early stages of software architecture design, when there is still some ambiguity in the problem description and software architecture design. Another cause is the lack of quality solution domain knowledge, which makes it difficult to derive relevant solution domain models for architectural modules [3]. When we are aware of the causes for accidental crosscutting it will be easier to identify inherent crosscutting. Additionally, we can recognize inherent crosscutting concerns from our own experience or experience of others.

Module Artifact Diagram

Figure 5.12 shows the module artifact diagram with the categories of modules used in COSAAM.

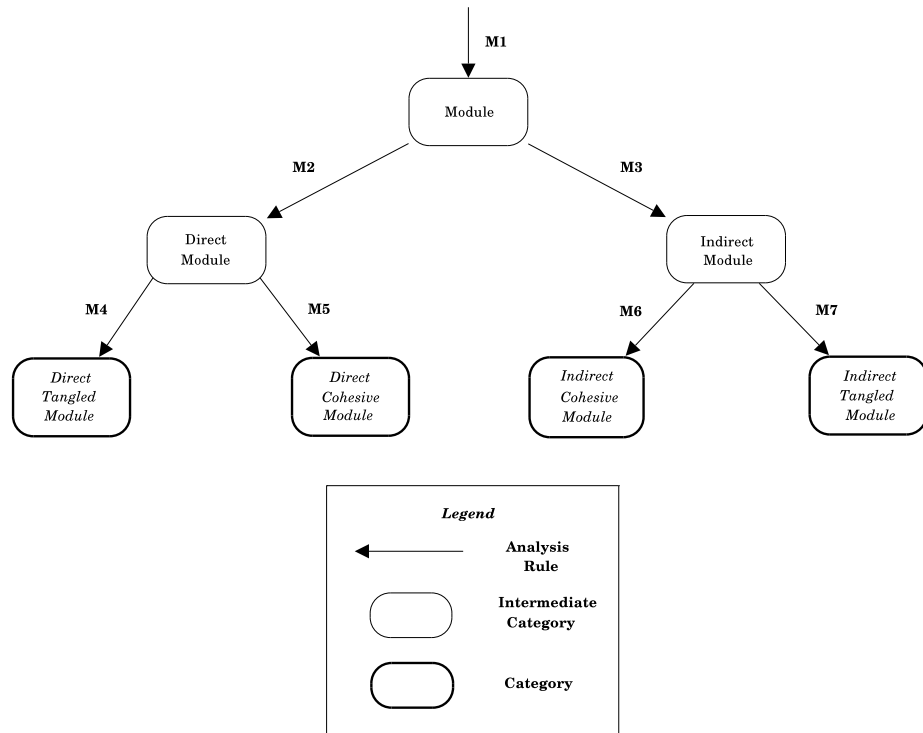


Figure 5.12: Module artifact diagram

Figure 5.13 shows the analysis rules for modules.

- M1:** SELECT module from preparation phase
- M2:** IF module addresses a concern directly
THEN module becomes a DIRECT MODULE
- M3:** IF module addresses a concern indirectly
THEN module becomes an INDIRECT MODULE
- M4:** IF DIRECT MODULE addresses multiple concerns
THEN module becomes a DIRECT TANGLED MODULE
- M5:** IF DIRECT MODULE addresses a single concern
THEN module becomes a DIRECT COHESIVE MODULE
- M6:** IF INDIRECT MODULE addresses a single concern
THEN module becomes an INDIRECT COHESIVE MODULE
- M7:** IF INDIRECT MODULE addresses multiple concerns
THEN module becomes an INDIRECT TANGLED MODULE

Figure 5.13: Module analysis rules

- *Direct Cohesive Module (DC)*

A *Direct Cohesive Module* is a module that addresses a single concern directly. Figure 5.14 shows the DMM mapping that corresponds to the Direct Cohesive Module.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td></td><td></td></tr><tr><td>C1</td><td>D</td><td></td><td></td></tr><tr><td>C2</td><td></td><td></td><td></td></tr><tr><td>C3</td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td><td>DC</td><td></td><td></td></tr></table>		M1			C1	D			C2				C3				cat	DC			<div>Direct Cohesive Module (DC)</div>
	M1																				
C1	D																				
C2																					
C3																					
cat	DC																				

Figure 5.14: DMM mapping of *Direct Cohesive Module* (DC)

- *Indirect Cohesive Module (IC)*

An *Indirect Cohesive Module* is a module that addresses a single concern indirectly. Figure 5.15 shows a DMM mapping of the Indirect Cohesive Module.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td></td><td></td></tr><tr><td>C1</td><td>I</td><td></td><td></td></tr><tr><td>C2</td><td></td><td></td><td></td></tr><tr><td>C3</td><td></td><td></td><td></td></tr></table> <table><tr><td>cat</td><td>IC</td><td></td><td></td></tr></table>		M1			C1	I			C2				C3				cat	IC			<div>Indirect Cohesive Module (IC)</div>
	M1																				
C1	I																				
C2																					
C3																					
cat	IC																				

Figure 5.15: DMM mapping of *Indirect Cohesive Module (IC)*

- *Direct Tangled Module (DT)*

An *Direct Tangled Module* is a module that addresses multiple concerns directly. Figure 5.16 shows a DMM mapping of the Direct Tangled Module.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td></td><td></td></tr><tr><td>C1</td><td>D</td><td></td><td></td></tr><tr><td>C2</td><td>D</td><td></td><td></td></tr><tr><td>C3</td><td>D</td><td></td><td></td></tr></table> <table><tr><td>cat</td><td>DT</td><td></td><td></td></tr></table>		M1			C1	D			C2	D			C3	D			cat	DT			<div>Direct Tangled Module (DT)</div>
	M1																				
C1	D																				
C2	D																				
C3	D																				
cat	DT																				

Figure 5.16: DMM mapping of *Direct Tangled Module (DT)*

- *Indirect Tangled Module (IT)*

An *Indirect Tangled Module* is a module that addresses multiple concerns, of which at least one indirectly. Figure 5.17 shows a DMM mapping of the Indirect Tangled Module.

<i>DMM Mapping</i>	<i>Category</i>																				
<table><tr><td></td><td>M1</td><td></td><td></td></tr><tr><td>C1</td><td>I</td><td></td><td></td></tr><tr><td>C2</td><td>I</td><td></td><td></td></tr><tr><td>C3</td><td>I</td><td></td><td></td></tr></table> <table><tr><td>cat</td><td>IT</td><td></td><td></td></tr></table>		M1			C1	I			C2	I			C3	I			cat	IT			<div>Indirect Tangled Module (IT)</div>
	M1																				
C1	I																				
C2	I																				
C3	I																				
cat	IT																				

Figure 5.17: DMM mapping of *Indirect Tangled Module* (IT)

Category Summary

Figure 5.18 summarizes all categories of concerns.

<i>ID</i>	<i>Category</i>
N	New Concern
DL	Direct Local Concern
IL	Indirect Local Concern
DS	Direct Scattered Concern
IS	Indirect Scattered Concern
DX	Direct Crosscutting Concern
IX	Indirect Crosscutting Concern

Figure 5.18: Concern categories for the characterization of the mapping between concerns and modules

Figure 5.19 provides a summary of all module categories of the characterization of the mapping between concerns and modules.

<i>ID</i>	<i>Category</i>
DC	Direct Cohesive Module
I	Indirect Cohesive Module
DT	Direct Tangled Module
IT	Indirect Tangled Module

Figure 5.19: Module categories for the characterization of the mapping between concerns and modules

In the next section, the mappings between concerns and modules of the window manager software architecture are characterized. Figure 5.20 shows the concern-module DMM with additional columns and rows for the category of each concern and module.

	EM	WM	PM	SM	cat
MO	I	I	I	I	IX
OP	I	I	I	I	IX
FM	I	I	I	I	IX
PT	D	D	D		DS
PM		D	D		DS
IDM	I				IL
WM		D			DL
WAC		D			DL
SM				D	DL

cat	IT	IT	IT	IT
-----	-----------	-----------	-----------	-----------

Figure 5.20: DMM of window manager software architecture with the characterization of concerns and modules

We demonstrate the reasoning behind the characterization of concerns and modules for each of the categories. We mention which analysis rules are applied

and why.

Direct Local Concerns

The three concerns *Screen Management*, *Window Management* and *Window Appearance Configuration* are *Direct Local Concerns*. *Screen Management* is addressed by the *Screen Manager*. The *Window Management* and *Window Appearance Management* concerns all involve manipulation of windows and are addressed by the *Window Manager* (C2).

Indirect Local Concerns

The only *Indirect Local Concern* is *Input Device Management*. This concern originates from the requirement to use new devices for interacting with the window manager, such as touch screens and light pens. This concern requires changes to the *Event Manager* module, since the input devices have to be detected by the *Event Manager*. Because of these changes, *Input Device Management* is an *Indirect Local Concern* (C3).

Direct Scattered Concerns and Indirect Crosscutting Concerns

The concerns *Monitoring*, *Operating Systems Portability*, *Process Termination*, *Failure Management* and *Process Management* are *Indirect Crosscutting Concerns*, (C4, C8 and C10). However, we need to investigate if they are inherent crosscutting concerns. *Monitoring* is a well-known example of a crosscutting concern in literature [16], [25], [22]. *Failure Management* represents the management of failures of individual modules. Coordination of system-wide failure policies is an inherently crosscutting problem. *Operating Systems Portability* represents the selection of platform-specific components. This concern is relevant for every module that uses platform-specific components. Because of these reasons, we identify these three concerns as actual *Indirect Crosscutting Concerns* (C10). The concerns *Process Termination* and *Process Management* are not inherently crosscutting because process related activities are relatively isolated from other activities in the window manager. As a result, they are *Direct Scattered Concerns* (C11).

Indirect Tangled Modules

All modules in the window manager software architecture are *Indirect Tangled Modules* because they address multiple concerns and because the existence of *Indirect Crosscutting Concerns*. Note that the number of concerns is more than the double the number of modules. Whenever there are more concerns than modules in a DMM, and they are all directly or indirectly mapped to these modules, there will be tangling.

5.1.3 Activity: Measure Scattering and Tangling

- *Goal*

The goal of this activity is to measure scattering of concerns and tangling of modules, based on the concern-module DMM.

- *Inputs*

The input of this activity is the concern-module DMM.

- *Outputs*

The output of this activity is a collection of metrics for scattering and tangling calculated from the concern-module DMM.

In the previous activity we have characterized the mapping between the concerns and modules of the window manager software architecture. In this activity we measure scattering of concerns and tangling of modules, based on the mappings in the concern-module DMM. These metrics are used to show the severity of problems in the existing software architecture design. Additionally, scattering and tangling are used to control the evaluation process. The goal of the transformation is to systematically reduce scattering and tangling in the software architecture, while addressing the concerns of stakeholders by the modules in the software architecture. This explicit reasoning is applied during the transformation phase to determine when and how the software architecture should be transformed.

We present three metrics to measure scattering and tangling of concerns and modules in the concern-module DMM. These metrics represent the severity of scattering and tangling in the software architecture. In existing literature on aspect-oriented refactoring, a scattering degree metric has been used to quantify aspects in middleware [47]. Figueiredo et. al. present a way towards a quantitative method for assessing aspect-oriented artifacts based on implementation metrics [15]. COSAAM adopts a similar approach and applies measures of scattering and tangling degrees at the architecture level. The measure of scattering and tangling is less precise at the architecture level, since architectural concerns and modules are conceptual in nature, as opposed to implementation units. Nevertheless, these metrics are still useful to determine the severity of scattering and tangling in software architectures. These approaches can be combined to investigate software architectures in combination with round-trip engineering, which connects software architecture metrics with implementation metrics. We distinguish between *direct* and *indirect* scattering and tangling degrees for concerns and modules, because this distinction influences the transformation process.

5.1.4 Concern Metrics

The concern-module DMM consists of n concerns and m modules, C_0 up to including C_n and M_0 up to including M_m . Figure 5.21 shows three concern

metrics for the concern-module DMM.

- **Direct Scattering Degree - dsd**

- *Measure*

- The direct scattering degree measures how many modules in the current candidate software architecture are responsible for addressing this concern.

- *Definition*

- Number of direct mappings concern C_i is part of.

- *Symbol*

- $dsd(C_i)$

- **Indirect Scattering Degree - isd**

- *Measure*

- The indirect scattering degree measures how many modules in the candidate software architecture have to be modified in order to address this concern.

- *Definition*

- Number of indirect mappings concern C_i is part of.

- *Symbol*

- $isd(C_i)$

- **Scattering Degree - sd**

- *Measure*

- The scattering degree measures how many modules directly or indirectly address this concern. It is the sum of the values of the direct and indirect scattering degree.

- *Definition*

- Number of direct and indirect mappings concern C_i is part of.

- *Symbol*

- $sd(C_i) = dsd(C_i) + isd(C_i)$

Figure 5.21: Scattering metrics for concerns

There are also three tangling metrics for modules. These are similar as the three metrics above. Figure 5.22 shows the tangling metrics of modules based on the concern-module DMM.

- **Direct Tangling Degree - dtd**

- *Measure*

- The direct tangling degree measures how many concerns are directly addressed by this module.

- *Definition*

- Number of direct mappings module M_j is part of.

- *Symbol*

- $dtd(M_j)$

- **Indirect Tangling Degree - itd**

- *Measure*

- The indirect tangling degree measures how many concerns are indirectly addressed by this module in the candidate software architecture.

- *Definition*

- Number of indirect mappings module M_j is part of.

- *Symbol*

- $itd(M_j)$

- **Tangling Degree - td**

- *Measure*

- The tangling degree measures how many concerns are directly or indirectly addressed by this module.

- *Definition*

- Number of direct and indirect mappings module M_j is part of.

- *Symbol*

- $td(M_j) = dtd(M_j) + itd(M_j)$

Figure 5.22: Tangling metrics for modules

We will determine the values of these metrics for the window manager software architecture. Figure 5.23 shows the DMM and all metrics values.

	EM	WM	PM	SM		dsd	isd	sd	cat
MO	I	I	I	I		0	4	4	IX
OP	I	I	I	I		0	4	4	IX
FM	I	I	I	I		0	4	4	IX
PT	D	D	D			3	0	3	DS
PM		D	D			2	0	2	DS
IDM	I					0	1	1	IL
WM		D				1	0	1	DL
WAC		D				1	0	1	DL
SM				D		1	0	1	DL

dtd	1	4	2	1
itd	4	3	3	3
td	5	7	5	4
cat	IT	IT	IT	IT

Figure 5.23: Concern-module DMM with concern and module categories and scattering and tangling metrics

The concern-module DMM shows that the *Monitoring*, *OS Portability* and *Failure Management* concerns have the highest *isd* and *sd* values of 4. Additionally, the Window Manager module has the highest *tmd* value of 6. This means that the *Window Manager* is the least cohesive module, with respect to the number of concerns has to address. This high value denotes that explicit reasoning about its responsibilities will be relatively difficult. The evolution of this module can be problematic, since all of the concerns it should address can evolve individually. Note that there are more metrics that could be defined for the concern-module DMM. For example, for prioritization of concerns, we could define the weight of each concern by counting the number of scenarios that are in the cluster that corresponds to that concern. This metric is not a part of COSAAM but this could be applied in future work in this area.

Chapter 6

COSAAM Transformation Phase

This chapter describes the activities of the COSAAM transformation phase. Section 6.1 provides a short introduction to the concept of software architecture transformation. Section 6.2 provides an overview of the activities of the transformation phase. Sections 6.3, 6.4 and 6.5 describe the *Initialize and Sequence Architecture DSM*, *Select Transformation Rule* and *Apply Transformation Rule* activities respectively.

6.1 Software Architecture Transformation

Software transformation can be broadly defined as the process of changing software architectures. Any time the software architecture is changed from one version to the next, it has been transformed. Software architecture transformation is a combination of modification, refactoring and design. In the following paragraphs we provide a background for each of these elements.

Architecture modification is changing the externally observable behavior of the software architecture design. This includes defining or expanding new features or an increase of the scope of requirements. Modification changes the behavior of the software architecture and consists of minor changes of its internal structure.

Refactoring is a programming technique that is used to increase the quality of source code without changing its externally observable behavior [17]. Since several years, the concept of refactoring has been applied at the level of software architectures. Refactoring reduces scattering and tangling of code fragments and as a result, the implementation becomes easier to modify and maintain.

Research on refactoring indicates that it is important to determining *where* and *when* to refactor [8]. In code refactoring, software metrics are used to

locate areas that require refactoring. These metrics measure size, complexity, cohesion, coupling [8]. COSAAM adopts a similar approach to measure and reduce scattering and tangling, based on metrics for the concern-module DMM. A benefit of these metrics is that we can measure progress during the evaluation. Other existing refactoring approaches are based on visualization of underlying structure of the software. To determine *when* to refactor, we need to choose a sequence of individual refactorings. This prevents conflicts and the so-called "ripple-effect" that may result from change propagation [8]. To cope with these problems, COSAAM should provide heuristics to sequence transformations of the software architecture.

6.2 Transformation Phase Overview

The COSAAM transformation phase aims to address the problems associated with architecture transformation, described in the previous section. Figure 6.1 shows an UML activity diagram for the activities in the COSAAM transformation phase. The transformation phase consists of the activities *Initialize and Sequence Architecture DSM*, *Select Transformation Rule* and *Apply Transformation Rule* that are performed sequentially. For each activity we describe its goal, inputs and outputs and apply the activity on the window manager software architecture.

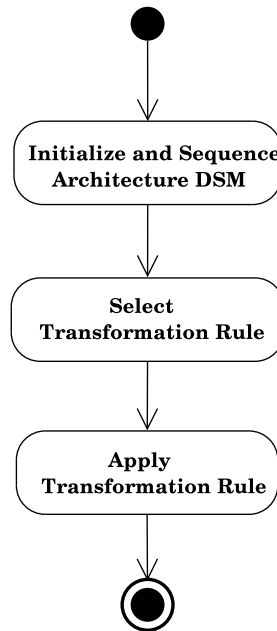


Figure 6.1: UML activity diagram of the activities of the COSAAM transformation phase

6.3 Activity: Initialize and Sequence Architecture DSM

- *Goal*

The goal of this activity is to sequence the architecture DSM to help define an ordering of possible transformations.

- *Inputs*

The input of this activity is the candidate software architecture.

- *Outputs*

The output of this activity is a DSM of the modules and the relationships of the candidate software architecture. The elements of the DSM are the modules of the architecture and the cells contain module relationships. The DSM is sequenced with respect to the dependencies caused by module relationships.

Rework is caused by changes in information that is required to perform an activity [10]. Since software architecture transformation changes the modules of a software architecture, there is the possibility of rework of designing modules during the process. The transformation of a module requires information about interfaces of the modules it depends on. When these interfaces change, the module that uses these interfaces may have to be design or transformed again. To cope with this problem, a DSM representation of the software architecture is used to view dependencies between modules. By examining the dependencies we can prevent unnecessary work during the transformation of the candidate software architecture. The following sections explain how to create this DSM representation.

6.3.1 DSM Representation of Module Relationships

This section shows how modules and their relationships are represented in the DSM. While there may be many types of module relationships, only four specific relationships are examined. These are the *Usage*, *Composition*, *Specialization* and *Advice* relationships. All of these relationships have a unique representation in the DSM. After we have described the representation of each module relationship in the DSM we show how to initialize the architecture DSM for the window manager software architecture. The following subsections show how each module relationship is represented in both the software architecture and the DSM:

Usage Relationships

A usage relationship between module M1 and module M2 means that M1 uses the services of M2. Because of this, module M1 is dependent on module M2, since a change in the interface or services of module M2 can affect module

M1. Figure 6.2 shows the representation of the usage relationship in both the software architecture and the DSM. The DSM in this figure is read as follows: a module on a column is dependent on a module on a row if there is an icon of a module relationship in the corresponding cell. The DSM shows all dependencies downwards, in a uniform manner.

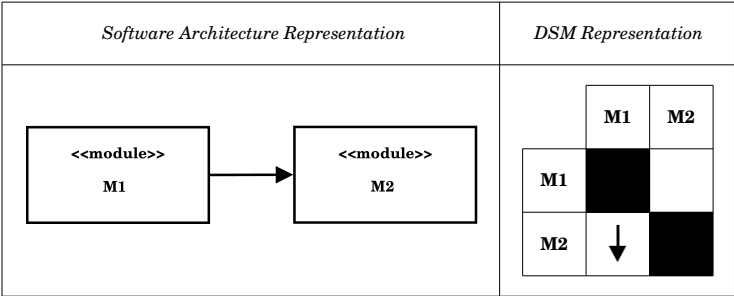


Figure 6.2: Representation of a usage relationship in a software architecture design and DSM

Composition Relationship

A composition relationship is similar to a usage relationship, but the main difference is that module M2 is a part of module M1. Because of this, module M1 is dependent on module M2. Figure 6.3 shows the representation of the composition relationship in both the software architecture and the DSM.

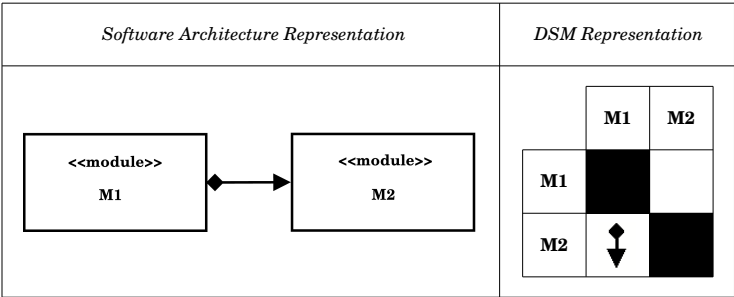


Figure 6.3: Representation of a composition relationship in a software architecture design and DSM

Specialization Relationship

A specialization relationship between module M1 and module M2 means that module M1 is a special instance of the generic module M2. Because the specialized module M1 is dependent on the interface or services of a generic module M2, a change in M2 changes module M1. For this reason, the

specialization arrow is pointing downwards in the DSM, instead of upwards — the way it is often represented in UML diagrams [18]. Figure 6.4 shows the representation of the specialization relationship in both the software architecture and the DSM.

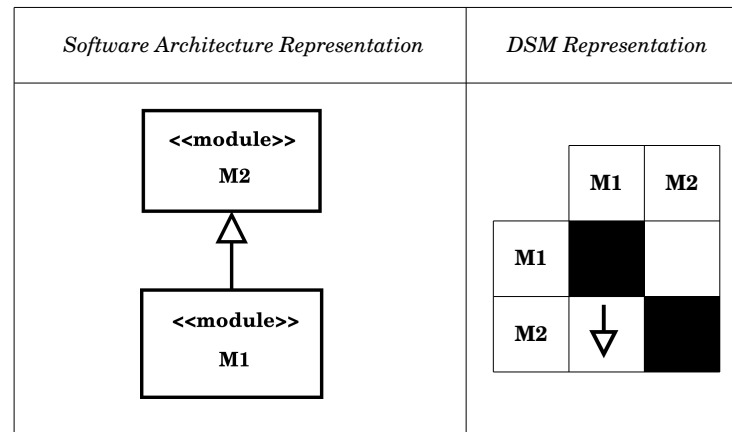


Figure 6.4: Representation of a specialization relationships in a software architecture design and DSM

Advice Relationship

An advice relationship between module M1 and M2 means that module M1 influences the behavior of module M2 through aspect-oriented composition techniques. Module M2 is, in theory, oblivious to the influence of module M1. Module M1, on the other hand is dependent on the interface of module M2. If this module would be implemented, aspect-oriented composition mechanisms would have to be defined, based on the interface of M2. Examples of aspect-oriented composition mechanisms are AspectJ pointcut specifications or composition filters declarations in Compose* [1], [16]. Figure 6.5 shows the representation of the advice relationship in both the software architecture and the DSM.

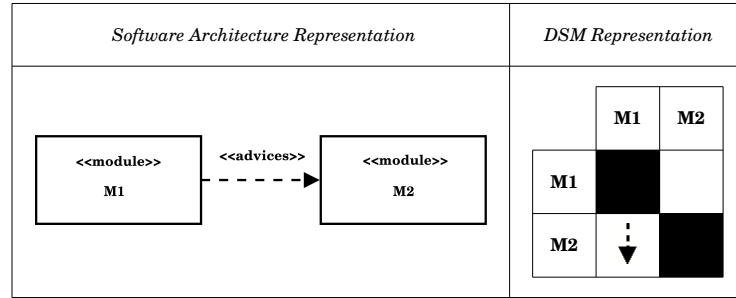


Figure 6.5: Representation of an advice relationship in a software architecture design and DSM

6.3.2 Architecture DSM for the Window Manager Software Architecture

Figure 6.6 shows the module view of the window manager software architecture. The figure contains modules and their relationships. The window manager software architecture has four modules and three usage relationships between modules.

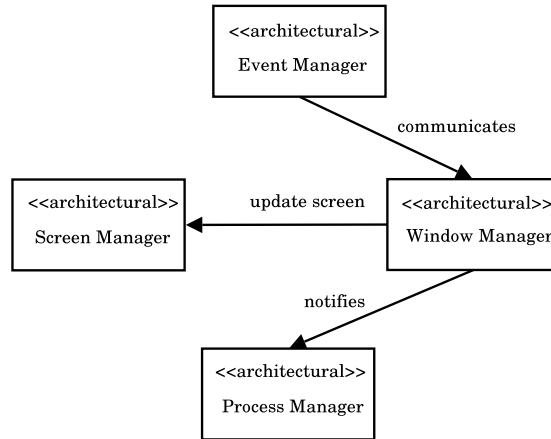


Figure 6.6: Window Manager Software Architecture

The DSM is initialized using the examples provided by previous figures. The architecture diagram shows that the Event Manager has a *usage relationship* with the Window Manager. The Event Manager sends updates from user and system events to the Window Manager. This relationship is represented in the DSM by putting a usage icon in the cell that corresponds to the Event Manager (EM) column and Window Manager (WM) row. The window manager itself uses the Process Manager (PM) and Screen Manager (SM) and two usage icons

have been added in appropriate cells. Figure 6.7 shows the complete DSM representation of the window manager software architecture.

	EM	PM	WM	SM
EM				
PM			↓	
WM	↓			
SM			↓	

Figure 6.7: DSM representation of window manager software architecture

A sequenced DSM provides useful information for managing processes and dependencies between activities. In COSAAM a sequenced architecture DSM is used to help to determine the ordering in which modules and concerns are transformed. Additionally, the DSM is used minimize rework caused by dependencies between modules. The architecture DSM is transformed into a lower-triangular form by topologically sorting its graph representation. The topological sorting algorithm is well-known graph algorithm [13]. The algorithm is used to sort the modules according to the sequence of dependencies with other modules. The lower-triangular form helps to reason about dependencies between modules and dependencies between transformations. Figure 6.8 shows the DSM of the window manager software architecture in lower-triangular form.

	EM	WM	PM	SM
EM				
WM	↓			
PM		↓		
SM		↓		

Figure 6.8: DSM representation of window manager software architecture in lower-triangular form after sequencing

6.4 Activity: Select Transformation Rule

- *Goal*

The goal of this activity is to select an appropriate transformation rule to transform the software architecture.

- *Inputs*

The inputs of this activity are a sequenced architecture DSM, a characterization of concerns and modules in the concern-module DMM and measures of scattering and tangling.

- *Outputs*

The output of this activity is the selected transformation rule.

The activities of the analysis phase provide a characterization of concerns and modules and measures of scattering and tangling of concerns and modules. Additionally, a sequenced architecture DSM is provided that shows dependencies between modules. In this activity, these elements are combined with a set of heuristics to select an appropriate transformation rule. Before transformation rules are discussed several primitive transformations of the DMM and DSM are defined. Then the transformation rules are expressed in terms of primitive matrix transformations. Furthermore, heuristics are defined for selecting a particular transformation rule to transform the software architecture. Finally, a transformation rule is selected that will be applied to the window manager software architecture in the next activity, *Apply Transformation Rule*.

6.4.1 Primitive DMM and DSM Transformations

At the start of this activity, the DMM and DSM are both initialized. We define a collection of primitive transformations for both the DMM and DSM. These primitive transformation form the basis of more complex transformation rules that we define in the next section.

Figure 6.9 shows the transformation for the concern-module DMM and their effects.

Primitive DMM Transformation	Effect on DMM
Add concern	Add concern in a new row
Remove concern	Remove concern in a row
Add module	Add module in new column
Remove module	Remove module in a column
Add Direct Mapping	Add D to a cell
Remove Direct Mapping	Remove D from a cell
Add Indirect Mapping	Add I to a cell
Remove Indirect Mapping	Remove I from a cell

Figure 6.9: Primitive concern-module DMM transformations and their effects

Similar transformations are defined for the architecture DSM. Figure 6.10 shows each primitive transformation for the architecture DSM and their effects. Note that some transformations have an effect on both the concern-module DMM and the architecture DSM.

Primitive DSM Transformation	Effect on DSM
Add module	Add module in new row and column
Remove module	Remove module in row and column
Add usage relationship	Add usage relationship in cell
Remove usage relationship	Remove usage relationship from a cell
Add specialization relationship	Add specialization relationship in a cell
Remove specialization relationship	Remove specialization relationship from a cell
Add composition relationship	Add composition relationship in a cell
Remove composition relationship	Remove usage relationship from a cell
Add advice relationship	Add specialization relationship in a cell
Remove advice relationship	Remove usage relationship from a cell

Figure 6.10: Primitive architecture DSM transformations and their effects

With these primitive transformations, any kind of configuration in the DMM and DSM can be made. For the purpose of transforming software architectures, the transformations that reduce scattering and tangling in the candidate architecture and are interesting. In the next section several transformation rules are defined for this purpose, which can be expressed as a composition of primitive DMM and DSM transformations.

6.4.2 Transformation Rules

This section defines transformation rules which improve the software architecture and reduce or eliminate scattering and tangling of concerns and modules. To reduce scattering and tangling the number of *Direct Local Concerns* and the number of *Direct Cohesive Modules* has to increase. During this process the amount of rework due to module dependencies has to be minimized.

Concern Transformation Rules

Figure 6.11 shows the concern artifact diagram from the analysis phase in chapter 5 with added transformation rules. The transformation rules transform a concern of a certain category to another concern category, reducing scattering or tangling in the process. The ultimate goal is to transform every concern into a *Direct Local Concern*

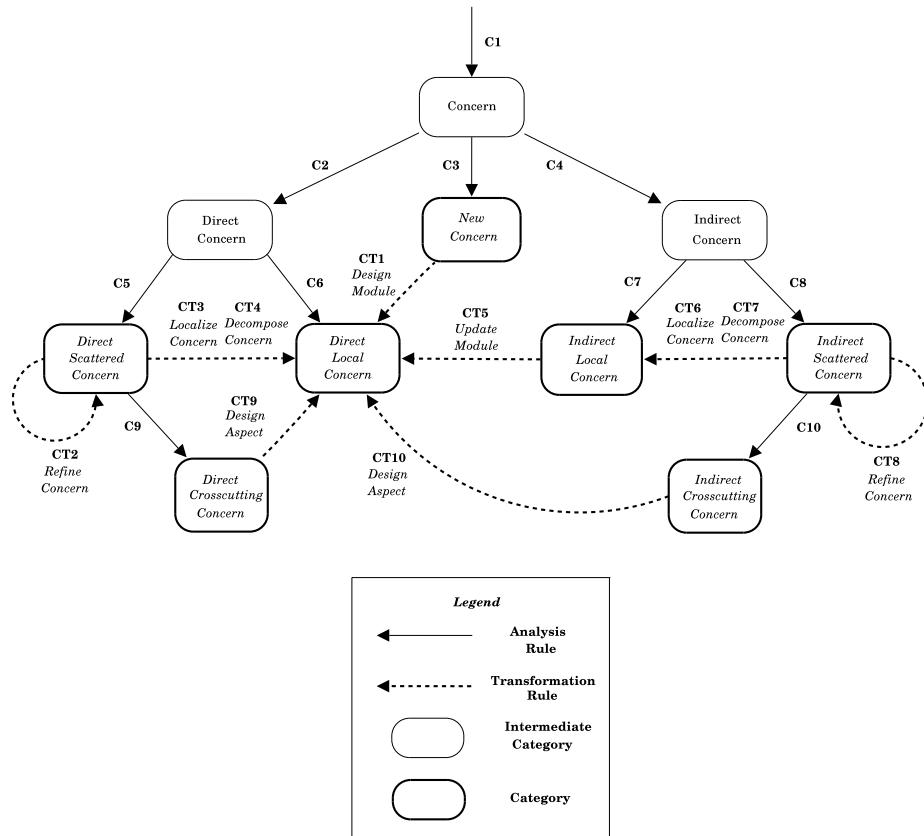


Figure 6.11: Concern artifact diagram with added transformation rules

Figure 6.12 shows the transformation rules for concerns.

ID	Transformation Rule	Predicate
CT1:	Design Module	IF NEW CONCEN can be addressed by newly designed module THEN concern becomes a DIRECT LOCAL CONCERN
CT2:	Refine Concern	IF DIRECT SCATTERED CONCERN can be split into distinct concerns from which at least one concern is directly addressed by a single module THEN concern becomes a DIRECT SCATTERED CONCERN
CT3:	Localize Concern	IF DIRECT SCATTERED CONCERN can be localized in a single module THEN concern becomes a DIRECT LOCAL CONCERN
CT4:	Decompose Concern	IF DIRECT SCATTERED CONCERN can be decomposed into distinct concerns which can all be directly addressed by separate modules THEN concern becomes a DIRECT LOCAL CONCERN
CT5:	Update Module	IF INDIRECT LOCAL CONCERN can be directly addressed after updating the responsibilities of the module THEN concern becomes a DIRECT LOCAL CONCERN
CT6:	Localize Concern	IF INDIRECT SCATTERED CONCERN can be indirectly addressed by a single module THEN concern becomes an INDIRECT LOCAL CONCERN
CT7:	Decompose Concern	IF INDIRECT SCATTERED CONCERN can be decomposed into distinct concerns which can all be indirectly addressed by separate modules THEN concern becomes an INDIRECT LOCAL CONCERN
CT8:	Refine Concern	IF INDIRECT SCATTERED CONCERN can be decomposed into distinct concerns from which one least one is indirectly addressed by a single module THEN concern becomes an INDIRECT LOCAL CONCERN
CT9:	Design Aspect	IF DIRECT CROSSCUTTING CONCERN can be directly addressed by an aspect THEN concern becomes a DIRECT LOCAL CONCERN
CT10:	Design Aspect	IF INDIRECT CROSSCUTTING CONCERN can be directly addressed by an aspect THEN concern becomes a DIRECT LOCAL CONCERN

Figure 6.12: Concern transformation rules

The following sections discuss the transformation rules for concerns. In the application of rule **CT1**, a new module is design to address a *New Concern*. It is a rule that requires software architecture design activities. Accordingly, software architects may proceed through several or all stages of a software architecture design method of choice. For example, one can apply Synbad, the synthesis based software architecture design method, to design an architectural module through synthesis of relevant solution domain models [42].

Rules **CT2**, **CT3** and **CT4** are transformation rules for *Direct Scattered Concerns*. In rule **CT2**, the concern is partially decomposed and it remains *Direct Scattered Concern*. However, a part of the concern is now addressed by a *Direct Local Concern*. The other two rules eliminate scattering and tangling and transform the *Direct Scattered Concern* into a *Direct Local Concern*. The rules **CT6**, **CT7** and **CT8** apply similar transformations, that transform *Indirect Scattered Concerns* into *Indirect Local Concerns*.

In the application of rule **CT5**, the module that indirectly addresses the *Indirect Local Concern* is updated. Its responsibilities are updated or extended to address the concern and the concern becomes a *Direct Local Concern*. This transformation is a modification and not a refactoring, because it changes the externally observable behavior of the software architecture design. We can see that several rules can be applied sequentially to transform a concern into a *Direct Local Concern*. For example, rule **CT7** can transform an *Indirect Scattered Concern* into an *Indirect Local Concern*. Rule **CT5** can then be applied to transform the *Indirect Local Concern* to a *Direct Local Concern*.

The remaining two rules **CT9** and **CT10** transformation rules transform *Direct Crosscutting Concerns* and *Indirect Crosscutting Concerns* into *Direct Local Concerns*. This involves the design of an architectural aspect. This transformation involves changing existing module relationships, such as usage, composition and specialization into to aspect-oriented composition mechanisms.

Concern Transformation Rules and Primitive DMM and DSM Transformations

We can express these transformation rules in terms of primitive DMM and DSM transformations. Figure 6.13 shows how the concern transformation rules are expressed in terms of primitive DMM and DSM transformations.

ID	Transformation Rule	Primitive DMM and DSM Transformations
CT1:	Design Module	Add Module, Add Direct Mapping Add/Remove Module Relationships
CT2:	Refine Concern	Remove Direct Mapping, Add Concern, Add Module, Add Direct Mapping Add/Remove Module Relationships
CT3:	Localize Concern	Remove Direct Mapping Add/Remove Module Relationships
CT4:	Decompose Concern	Remove Direct Mapping, Add Concern, Add Module, Add Direct Mapping Add/Remove Module Relationships
CT5:	Update Module	Remove Indirect Mapping, Add Direct Mapping Add/Remove Module Relationships
CT6:	Localize Concern	Remove Indirect Mapping Add/Remove Module Relationships
CT7:	Decompose Concern	Remove Indirect Mapping, Add Concern, Add Module, Add Indirect Mapping Add/Remove Module Relationships
CT8:	Refine Concern	Remove Indirect Mapping, Add Concern, Add Module, Add Indirect Mapping Add/Remove Module Relationships
CT9:	Design Aspect	Remove Direct Mapping, Add Module Add/Remove Module Relationships
CT10:	Design Aspect	Remove Indirect Mapping, Add Module Add/Remove Module Relationships

Figure 6.13: Concern transformation rules with associated primitive DMM and DSM transformations

Module Transformation Rules

We define similar transformation rules for modules. The transformation rules transform a module of a module category to a representation that results in less scattering or tangling. Figure 6.14 shows the module artifact diagram from the analysis phase in chapter 5 with added transformation rules.

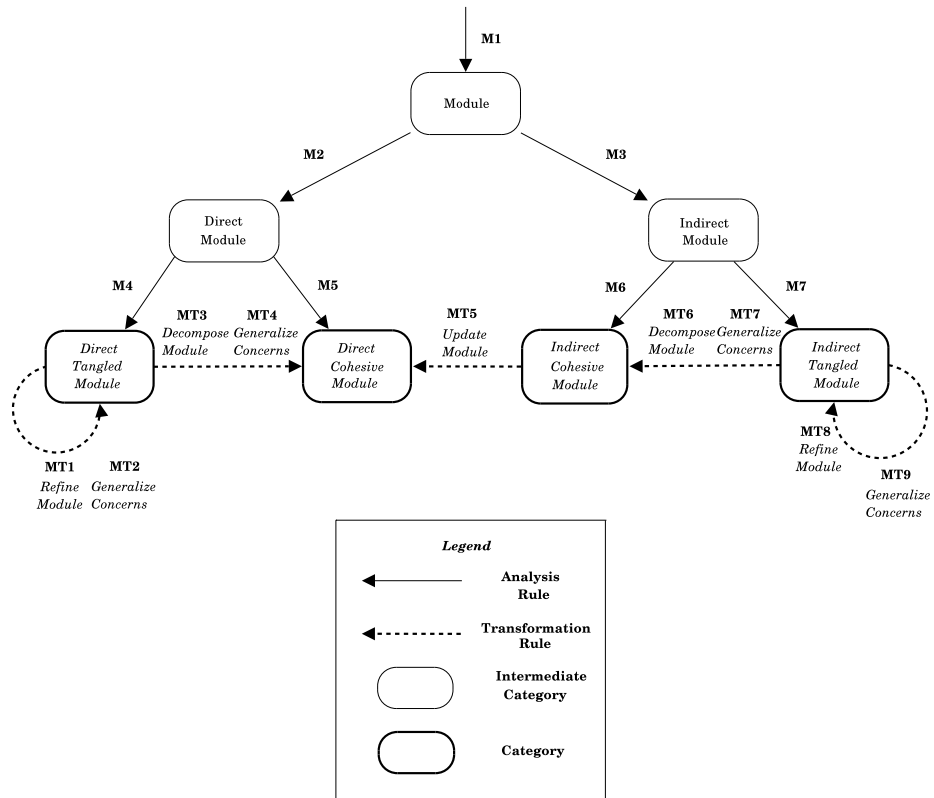


Figure 6.14: Module artifact diagram with added transformation rules

Figure 6.15 shows the transformation rules for modules.

ID	Transformation Rule	Predicate
MT1:	Refine Module	IF DIRECT TANGLED MODULE can be decomposed into distinct modules from which at least one module directly addresses a single concern THEN module becomes a DIRECT TANGLED MODULE
MT4:	Generalize Concerns	IF DIRECT TANGLED MODULE addresses concerns from which some can be generalized into a single concern THEN module becomes a DIRECT TANGLED MODULE
MT3:	Decompose Module	IF DIRECT TANGLED MODULE can be decomposed into distinct modules which all directly addresses a separate concern THEN module becomes a DIRECT COHESIVE MODULE
MT4:	Generalize Concerns	IF DIRECT TANGLED MODULE addresses concerns which all of them can be generalized into a single concern THEN module becomes a DIRECT COHESIVE MODULE
MT5:	Update Module	IF INDIRECT COHESIVE MODULE can be updated to directly address a single concern THEN concern becomes a DIRECT COHESIVE MODULE
MT6:	Decompose Module	IF INDIRECT TANGLED MODULE can be decomposed into distinct modules which all indirectly address a separate concern THEN module becomes an INDIRECT COHESIVE MODULE
MT7:	Generalize Concerns	IF INDIRECT TANGLED MODULE addresses concerns from which all of them can be generalized into a single concern THEN module becomes an INDIRECT COHESIVE MODULE
MT8:	Refine Module	IF INDIRECT TANGLED MODULE can be decomposed into distinct modules from which at least one module indirectly addresses a single concern THEN module becomes an INDIRECT TANGLED MODULE
MT9:	Generalize Concerns	IF INDIRECT TANGLED MODULE addresses concerns for which some can be generalized into a single concern THEN module becomes an INDIRECT TANGLED MODULE

Figure 6.15: Module transformation rules

The following sections elaborate on the transformation rules for modules. Rules **MT1** and **MT2** are transformation rules for *Direct Tangled Modules*. In rule **MT1**, the module is partially decomposed and it remains *Direct Tangled Module*. However, a part of original module now addresses a *Direct Local Concern*. The other two rules eliminate tangling and transform the *Direct Tangled Module* into a *Direct Cohesive Module*. The rules **MT4** and **MT5** apply similar transformations, that transform *Indirect Tangled Modules* into *Indirect Tangled Modules*.

In the application of rule **MT3**, the *Indirect Cohesive Module* is updated. Its responsibilities are updated or extended to address a single concern directly and becomes a *Direct Cohesive Module*. This transformation is the same as transformation rule **CT5** for concerns. We can also see that several rules can be applied sequentially to transform a concern into a *Direct Local Concern*. For example, rule **MT4** can transform an *Indirect Tangled Module* into an *Indirect Cohesive Module*. Rule **MT3** can then be applied to transform the *Indirect*

Cohesive Module into a *Direct Cohesive Module*.

6.4.3 Module Transformation Rules and Primitive DMM and DSM Transformations

We can express the module transformation rules in terms of primitive DMM and DSM transformations, like we did with the concern transformation rules. Figure 6.16 shows how the module transformation rules are expressed in terms of primitive DSM and DMM transformations.

ID	Transformation Rule	Predicate
MT1:	Refine Module	Remove Direct Mapping, Add Module, Add Direct Mapping Add/Remove Module Relationships
MT4:	Generalize Concerns	Remove Concern
MT3:	Decompose Module	Remove Indirect Mapping, Add Module Add Indirect Mapping
MT4:	Generalize Concerns	Remove Concern
MT5:	Update Module	Remove Indirect Mapping, Add Direct Mapping Add/Remove Module Relationships
MT6:	Decompose Module	Remove Indirect Mapping, Add Module, Add Direct Mapping Add/Remove Module Relationships
MT7:	Generalize Concerns	Remove Concern
MT8:	Refine Module	Remove Indirect Mapping, Add Module, Add Direct Mapping Add/Remove Module Relationships
MT9:	Generalize Concerns	Remove Concern

Figure 6.16: Module transformation rules with associated primitive DMM and DSM transformations

6.4.4 Heuristics for applying transformation rules

In the previous section we have defined *how* we can transform the software architecture. In this section we discuss *when* to apply the right transformations. We provide three heuristics that the evaluation team can use to select transformation rules. The evaluation team decides which transformation rule is most appropriate to apply.

1. *Defer the design of aspects*

The design of aspects introduces a phenomenon called *Dependency Inversion* [31]. When using object-oriented composition mechanisms, the dependencies of modules are aligned with the calling structure. With

aspect-oriented composition mechanisms, however, the aspect will depend on base modules. The dependency is inverted. The stable dependency principle states: ‘*Depend in the direction of stability*’ [28]. However, if these base modules are not direct modules, the aspect depends on an unstable basis. This can be problematic for modules that may be transformed later. Changes in base modules can propagate to aspects because of dependencies between aspect and base modules. This creates rework for redefining the aspect. Therefore, the design of aspects should be deferred to the point where the modules it cuts across are *Direct Cohesive Modules*. In case of multiple aspects, the evaluation team can decide which aspect to design first.

2. *Remove unnecessary mappings first*

The evaluation team should try to remove unnecessary mappings first. This can be done by applying the transformation rules *CT3: Localize Concern* and *MT3: Generalize Concerns*. This clears up the concern-module DMM and eases further transformation. This improves the knowledge of the software architecture to all people in the evaluation team. This is because participants discuss module responsibilities and the differences between concerns.

3. *Give priority to concerns and modules that are highly scattered or tangled*

The concerns and modules that exhibit the most scattering and tangling provide the greatest ambiguity in the software architecture design. As a result, the transforming these concerns or modules can have large impact on the architecture. These are concerns and modules with high *tcd* and *tmd* values. The uncertainty of this instability must be removed early to reduce ambiguity in the software architecture design and improve the accuracy of future transformation decisions.

6.4.5 Transforming the Window Manager Software Architecture

In this section we apply the activities we described in earlier sections to the window manager software architecture. Figure 6.17 shows the DMM at the start of the transformation phase. We now have several heuristics to help us select a transformation rule to transform the window manager software architecture. The sequenced architecture DSM shows that the Process Manager and Screen Manager do not depend on any other modules. They are located in the rightmost columns and lowest rows of the architecture DSM. The cells in these rows and columns do not contain any module relationships.

We choose to transform one of the concerns addressed by these modules. The Screen Manager has a *dcd* of 1, and is only in a mapping with crosscutting concerns. Therefore it requires no further transformation. We choose to transform the Process Manager. From the collection of concerns that are mapped to the Process Manager, we defer the transformation of the **Indirect**

Crosscutting Concerns (Heuristic 1). Instead, we choose to remove unnecessary mappings (Heuristic 2). This leaves us with a choice between two concerns: the Process Management and Process Termination Concern. We select the Process Termination concern first, since it has a high *dcd* degree value of 3 (Heuristic 3). Now that we have selected the concern to transform we decide to select transformation rule **CT3: Localize Concern**, to localize the Process Termination concern. In the next section we describe the activity of applying this transformation rule.

6.5 Activity: Apply Transformation Rule

During this activity, the selected transformation rule is performed and the DMM and DSMs are changed accordingly. The transformation is performed as an atomic operation. When the transformation is performed, a new iteration of the COSAAM activities can be done if necessary. Alternatively, stakeholders may decide to stop the process if scattering and tangling is eliminated or reduced to acceptable levels.

- *Goal*

The goal of this activity is to address concerns by the software architecture, reduce scattering and tangling of concerns and modules and minimize rework in future iterations.

- *Inputs*

The input of this activity is a transformation rule selected in the previous activity.

- *Outputs*

The output of this activity is a transformed software architecture.

	EM	WM	PM	SM		dsd	isd	sd	cat
MO	I	I	I	I		0	4	4	IX
OP	I	I	I	I		0	4	4	IX
FM	I	I	I	I		0	4	4	IX
PT	D	D	D			3	0	3	DS
PM		D	D			2	0	2	DS
IDM	I					0	1	1	IL
WP		D				1	0	1	DL
WAC		D				1	0	1	DL
SM				D		1	0	1	DL

dtd	1	4	2	1
itd	4	3	3	3
td	5	7	5	4
cat	T	T	T	T

Figure 6.17: DMM before localizing the *Process Termination* concern

The Process Termination concern is addressed by modules *Event Manager*, *Process Manager* and *Window Manager*. When we analyze how these modules cooperate we can conclude that terminating processes is actually a responsibility of the *Process Manager* only. The *Event Manager* provides events through the *Window Manager*, the *Window Manager* in turn communicates with the *Process Manager* to terminate a process. However, the actual act of terminating a process is the responsibility of the *Process Manager*. Therefore, we apply the selected transformation rule **CT3: Localize Concern** to transform the concern into a *Direct Local Concern* that is addressed by the *Process Manager*.

Figure 6.18 shows the DMM after the application of the transformation rule. We have shaded the row that has changed. The DSM has not been changed because we have not introduced any new modules or changed any module relationships. We have removed two direct mappings between the *Process Termination Concern* to the *Event Manager* and *Window Manager*. The

Process Termination concern is now addressed only by the Process Manager module. In the next iteration, the scattering and tangling degrees are calculated. However, to improve the readability of the case study, we combine the result of the application of the transformation rule with the characterization and calculation of metrics in the analysis phase that follows it. Figure 6.18 shows that *dsd* value of the *Process Termination* is now equal to 1.

	EM	WM	PM	SM	dmd	imd	tmd	cat
MO	I	I	I	I	0	4	4	X
OP	I	I	I	I	0	4	4	X
FM	I	I	I	I	0	4	4	X
PM		D	D		2	0	2	S
IDM	I				0	1	1	I
WP		D			1	0	1	D
WAC		D			1	0	1	D
PT			D		1	0	1	D
SM				D	1	0	1	D

dmd	0	3	2	1
imd	4	3	3	3
tmd	4	6	5	4
cat	T	T	T	T

Figure 6.18: DMM after localizing the *Process Termination* concern

We have described the goals, inputs and outputs of each activity in the transformation phase. At the end of this phase, the evaluation team has sequenced the architecture DSM, selected a transformation rule and applied it to the window manager software architecture and the DMM and DSM. The evaluation team can now perform another iteration, in which they select and apply a new transformation rule. After any iteration they decide to stop the evaluation, based on several stopping criteria, which are explained in the following section.

6.6 Stopping Criteria

At the end of the transformation phase, the evaluation team can decide to perform another iteration of COSAAM. Alternatively, they can stop further evaluation. We identify several stopping criteria that define when it may be better to stop analysis. These criteria can be situations of reaching a goal or meeting a problem that prevents the completion of the evaluation.

1. *Elimination of scattering and tangling*

This situation of elimination of scattering and tangling is characterized by the following necessary or sufficient conditions:

- All concerns are *Direct Local Concerns*. (necessary condition)
- All modules are *Direct Cohesive Modules*. (necessary condition)
- All *sd* and *td* values for concerns and modules are equal to 1. In this case, the DMM mapping is a bijection and there all scattering and tangling is eliminated. (sufficient condition)

2. *Acceptable reduction of scattering and tangling*

Alternatively, the levels of scattering and tangling may be acceptable and the evaluation is stopped. This depends on the prioritization of concerns and the opinions of the evaluation team.

3. *Insufficient Domain Knowledge*

Many transformation rules require appropriate problem- and solution domain knowledge, for defining concerns or decomposing and composing modules, respectively. If there is insufficient domain knowledge, it is difficult to proceed with the evaluation. A COSAAM evaluation may be stopped to start a search for domain knowledge. After appropriate knowledge is modeled the evaluation can proceed or restarted.

Chapter 7

Evolution of the Window Manager Software Architecture

In this chapter the transformation of the window manager software architecture is continued in eight iterations. After a transformation, the DMM and DSM are updated and during the activities of the analysis phase in the consecutive iteration the concern and modules categories and metrics are updated. To improve the readability of the description of these iterations, the changes in the transformation phase and the consecutive analysis phase are discussed together. The next section describes the second iteration of COSAAM.

7.1 Localizing the Process Management concern

Figure 7.1 shows the state of the DMM at the start of this iteration. The Process Management concern has the highest *dsd* of 3 and therefore needs to be transformed. The Process Management concern is transformed in the same way as in the previous iteration. The transformation rule **CT3: Localize Concern** is selected to make Process Management should be the responsibility of the Process Manager only. Figure 7.1 shows the result of the application of this transformation rule.

	EM	WM	PM	SM		dsd	isd	sd	cat
MO	I	I	I	I		0	4	4	IX
OP	I	I	I	I		0	4	4	IX
FM	I	I	I	I		0	4	4	IX
IDM	I					0	1	1	IL
WM		D				1	0	1	DL
WAC		D				1	0	1	DL
PM			D			1	0	1	DL
PT			D			1	0	1	DL
SM				D		1	0	1	DL

dtd	0	2	2	1
itd	4	3	3	3
td	4	5	5	4
cat	IT	IT	IT	IT

Figure 7.1: DMM after localizing the *Process Management* (PM) concern

7.2 Generalizing Process Termination and Process Management

7.2.1 Analysis & Transformation

The Process Manager addresses the concerns Process Management and Process Termination and has a *dtd* value of 2. There are two possible choices to reduce tangling in this module. The module can be decomposed (**MT6**) or its concerns can be generalized (**MT7**). If it is decomposed, two modules address each concern. In this case each module would address concerns that represent process

management activities, such as terminating, starting and interrupting processes. If its concerns are generalized however, the two concerns are merged into one concern. Generalizing the concerns maintains the simplicity of the design and is selected as a transformation. The choice between several transformation rules is based on background knowledge and experience of the evaluation team. Figure 7.2 shows the DMM after generalizing the concerns *Process Management* and *Process Termination*.

	EM	WM	PM	SM	dsd	isd	sd	cat
MO	I	I	I	I	0	4	4	IX
OP	I	I	I	I	0	4	4	IX
FM	I	I	I	I	0	4	4	IX
IDM	I				0	1	1	IL
WM		D			1	0	1	DL
WAC		D			1	0	1	DL
PM			D		1	0	1	DL
SM				D	1	0	1	DL

dtd	0	2	1	1
itd	4	3	3	3
td	4	5	4	4
cat	IT	IT	IT	IT

Figure 7.2: DMM after generalizing the concerns Process Management and Process Termination

7.3 Decomposing the Window Manager

The DMM from the previous iteration in figure 7.2 shows that the Screen Manager and the Process Manager have a *direct tangling degree* (*dtd*) value

of 1 and are tangled only due to the effect of crosscutting concerns. Since the design of aspects is deferred and they have a *dtd* value of 1 these modules do not require further transformation. The DSM shows that the next module to be transformed is the Window Manager, because it depends on the Process Manager and Screen Manager. The Window Manager has a *direct tangling degree* value of 2, because it addresses the two concerns Window Appearance Management and Window Management and therefore has to be transformed.

Similar to the previous iteration, there are two ways to reduce the tangling. The Window Manager can be decomposed (**MT6**) or its concerns can be generalized (**MT7**). Generalizing concerns would not be appropriate in this situation, since Window Management is a distinctively different concern than Window Appearance Management, which is the management of the visual appearance of windows. Therefore, the Window Manager is decomposed into two separate modules by applying transformation rule **MT6**. Figure 7.3 shows the effect of decomposing the Window Manager into a Window Manager and a Window Appearance Manager (WAM) on the DMM.

	EM	WM	WAM	PM	SM
MO	I	I	I	I	I
OP	I	I	I	I	I
FM	I	I	I	I	I
IDM	I				
WM		D			
WAC			D		
PM				D	
SM					D

dsd	isd	sd	cat
0	5	5	IX
0	5	5	IX
0	5	5	IX
0	1	1	IL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL

dtd	0	1	1	1	1
itd	4	3	3	3	3
td	4	4	4	4	4
cat	IT	IT	IT	IT	IT

Figure 7.3: DMM after decomposing the Window Manager

Notice that new mappings have been defined between the crosscutting concerns and the Window Appearance Manager. The concerns Operating Systems Portability, Monitoring and Failure Management have to be addressed by the new Window Appearance Manager as well. The Window Appearance Manager may use platform-specific components, such as specific window themes or plugins. In addition, activities of the Window Manager have to be monitored as well. Finally, failures that occur during its activities have to be managed. The definition of new mappings for these concerns indicates that these concerns are inherently crosscutting. Their scope evolves along with the introduction of new modules. The tangling caused by these crosscutting concerns turn the Window Appearance Manager into an *Indirect Tangled Module*. The reduction in tangling in one part of the software architecture leads to additional tangling, due

to the evolution of the scope of crosscutting concerns. This is a demonstration of the established notion that transformation with conventional composition mechanisms is not enough to deal with crosscutting [43].

This transformation affects the DSM, since we have introduced a new module. Figure 7.4 shows the DSM with the new Window Appearance Manager module. The Window Manager is composed the Window Appearance Manager through a composition relationship.

	EM	WM	WAM	PM	SM
EM					
WM	↓				
WAM		↕			
PM		↓			
SM		↓			

Figure 7.4: DSM with the new Window Appearance Manager module

The Window Appearance Manager is responsible for updating and loading settings for the visual appearance of windows. Examples of these are font settings, icons, themes, plug-ins et cetera. The Window Manager delegates changes to the appearance of windows to the Window Appearance Manager. The Window Manager’s main concern is the management of windows. The Window Manager is in theory oblivious to the appearance of the windows, which means that the appearance settings can be changed without affecting the Window Manager.

7.4 Defining the Event Management Concern

The DMM from figure 7.3 shows that the *direct tangling degree* of the Event Manager is 0 and indirectly addresses the Input Device Management concern. This means that it is currently not made explicit which concern is directly addressed by the Event Manager. Before the *Indirect Local Concern* Input Device Management can be addressed, the concern that is directly addressed by the Event Manager has to be made explicit. The mean responsibilities of

the Event Manager are capturing and transmitting events from the system or user to the Window Manager. Therefore a new concern is defined which is, unoriginally, called Event Management (EM). This small transformation might seem obvious. However, as the Window Manager evolves, the evaluation team gains more knowledge about the stakeholders' concerns and the responsibilities of modules. Even the obvious concern definitions need to be made explicit so that they can be changed or refined later, if necessary. Figure 7.5 shows the effect of the newly defined concern and mapping.

	EM	WM	WAM	PM	SM
MO	I	I	I	I	I
OP	I	I	I	I	I
FM	I	I	I	I	I
IDM	I				
EM	D				
WM		D			
WAC			D		
PM				D	
SM					D

dsd	isd	sd	cat
0	5	5	IX
0	5	5	IX
0	5	5	IX
0	1	1	IL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL

dtd	1	1	1	1	1
itd	4	3	3	3	3
td	5	4	4	4	4
cat	IT	IT	IT	IT	IT

Figure 7.5: DMM after defining an Event Management concern

7.4.1 Decomposing the Event Manager

As a result of the newly defined Event Management concern, the Event Manager now addresses the Event Management and Input Device Management concerns. There are again two transformation rules to reduce tangling of concerns in the Event Manager. The Event Manager can be decomposed (**MT6**) or its concerns can be generalized (**MT7**). Input Device Management involves registering input devices and assuring that events from different devices can be captured. Event Management on the other hand, involves communicating events, regardless of their source. Because these concerns are distinct and are likely to evolve separately, generalizing them would not be an option. Therefore, the Event Manager is decomposed into an Event Manager and an Input Device Manager (IDM). Figure 7.6 shows the effect of this transformation in the DMM.

	EM	IDM	WM	WAM	PM	SM		dsd	isd	sd	cat
MO	I	I	I	I	I	I		0	6	6	IX
OP	I	I	I	I	I	I		0	6	6	IX
FM	I	I	I	I	I	I		0	6	6	IX
EM	D							1	0	1	DL
IDM		D						1	0	1	IL
WM			D					1	0	1	DL
WAC				D				1	0	1	DL
PM					D			1	0	1	DL
SM						D		1	0	1	DL

dtd	1	1	1	1	1	1
itd	3	3	3	3	3	3
td	4	4	4	4	4	4
cat	IT	IT	IT	IT	IT	IT

Figure 7.6: DMM after decomposing the Event Manager with new Input Device Manager

Figure 7.7 shows the DSM with the new Input Device Manager module. A composition relationship has been defined between the Event Manager and the Input Device Manager.

	EM	IDM	WM	WAM	PM	SM
EM						
IDM	↕					
WM	↓					
WAM			↕			
PM			↓			
SM			↓			

Figure 7.7: DSM with the new Input Device Manager (IDM)

7.5 Designing an Operating System Bridge Aspect

In this iteration, only the crosscutting concerns require transformation. There is a choice between the crosscutting concerns *Monitoring*, *Failure Management* and *Operating Systems Portability*. It is decided to address the Operating Systems Portability concern by a newly defined Operating Systems Bridge aspect. The Operating Systems Bridge is an application of the *Bridge* design pattern, which helps to separate interface from implementation [19]. The Operating Systems Bridge aspect allows other modules to depend on platform independent components. To reduce the dependency on platform dependent components, the OS Bridge transparently redirects calls to virtual platform independent components to the appropriate platform dependent components. Examples of platform dependent components are program plug-ins, fonts, icons, windows, themes and so on. This way, the window manager can become portable throughout different operating systems. The Operating Systems Bridge can be configured centrally by determining how certain calls should be redirected, without changing other modules in the architecture. Figure 7.8 shows the effect of this transformation on the DMM.

	OB	EM	IDM	WM	WAM	PM	SM		dsd	isd	sd	cat
MO	I	I	I	I	I	I	I		0	7	7	IX
FM	I	I	I	I	I	I	I		0	7	7	IX
OP	D								1	0	1	DL
EM		D							1	0	1	DL
IDM			D						1	0	1	DL
WM				D					1	0	1	DL
WAC					D				1	0	1	DL
PM						D			1	0	1	DL
SM							D		1	0	1	DL

dtd	1	1	1	1	1	1	1
itd	2	2	2	2	2	2	2
td	3	3	3	3	3	3	3
cat	IT	IT	IT	IT	IT	IT	IT

Figure 7.8: DMM after the design of the Operating Systems Bridge aspect

Figure 7.11 shows the DSM with the new Operating Systems Bridge aspect.

	OB	EM	IDM	WM	WAM	PM	SM
OB							
EM	↓						
IDM	↓	↕					
WM	↓	↓					
WAM	↓			↕			
PM	↓			↓			
SM	↓			↓			

Figure 7.9: DSM after iteration seven - Map Operating Systems Portability concern to the Operating Systems Bridge Aspect

7.6 Designing a Failure Management Aspect

There are two remaining crosscutting concerns that have to be solved by the software architecture: the *Monitoring* and the *Failure Management* concern. In this iteration a Failure Management aspect is designed, which addresses the Failure Management concern. The Failure Manager provides a recovery mechanism to recover from failures in several modules. A detailed investigation of this concern should define detailed advice definitions and compositions. This is however beyond the scope of the case study. Figure 7.10 shows the DMM after this transformation.

	FM	OB	EM	IDM	WM	WAM	PM	SM
MO	I	I	I	I	I	I	I	I
FM	D							
OP		D						
EM			D					
IDM				D				
WM					D			
WAC						D		
PM							D	
SM								D

dsd	isd	sd	cat
8	0	8	IX
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL

dtd	1	1	1	1	1	1	1	1
itd	1	1	1	1	1	1	1	1
td	2	2	2	2	2	2	2	2
cat	IT	IT	IT	IT	IT	IT	IT	IT

Figure 7.10: DMM after the design of the Failure Management Aspect

Figure 7.11 shows the DSM with the new Failure Management Aspect.

	FM	OB	EM	IDM	WM	WAM	PM	SM
FM								
OB	↓							
EM	↓	↓						
IDM	↓	↓	↓					
WM	↓	↓	↓					
WAM	↓	↓			↓			
PM	↓	↓			↓			
SM	↓	↓			↓			

Figure 7.11: DSM after the design of the Failure Management Aspect

7.7 Designing a new Monitoring aspect

The final crosscutting concern to be addressed is the Monitoring concern. This iteration describes the design of a Monitoring aspect for this purpose. The monitor aspect monitors the activities of the user. For this reason we have to make sure to define advice relationships on all modules, so that each module activity can be logged. The exact workings and nuances of the monitor aspect should be defined during detailed design and implementation. Examples of things to consider are the definition of filters on the monitor log and detection of patterns of activity instead of single activities. Figure 7.12 shows DMM after iteration nine. The leftmost column shows the monitoring aspect. These requirements may lead to the definition of new concerns that can be used in a new iteration of COSAAM.

	MO	FM	OB	EM	IDM	WM	WAM	PM	SM
MO	D								
FM		D							
OP			D						
EM				D					
IDM					D				
WM						D			
WAC							D		
PM								D	
SM									D

dsd	isd	sd	cat
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL
1	0	1	DL

dtd	1	1	1	1	1	1	1	1	1
itd	0	0	0	0	0	0	0	0	0
td	1	1	1	1	1	1	1	1	1
cat	DC	DC	DC	DC	DC	DC	DC	DC	DC

Figure 7.12: DMM after the design of the Monitoring Aspect

The DSM has also changed in this iteration. Figure 7.13 shows the DSM with the new monitoring module. The module is an upper layer since no other modules depend on it. Additionally, it advises all other modules. This can be seen in the first column.

	MO	FM	OB	EM	IDM	WM	WAM	PM	SM
MO									
FM	↓								
OB	↓	↓							
EM	↓	↓	↓						
IDM	↓	↓	↓	↕					
WM	↓	↓	↓	↓					
WAM	↓	↓	↓			↕			
PM	↓	↓	↓			↓			
SM	↓	↓	↓			↓			

Figure 7.13: The new *Monitoring* module, (MO), advises all other modules

Figure 7.14 shows the UML representation of the transformed window manager software architecture. Like in the DSM the top of the figure shows the aspects and the bottom of the figure shows the modules. The modules are composed with conventional *usage* and *composition* relationships. The aspects are composed with each other with *advice* relationships. Additionally, the aspects advice all modules below. We have not drawn all individual advice relationships, as this would clutter the diagram with crossed lines from the advice relationships. Instead, we have created a box around all the module and attached the *advice* relationships from the aspects to the box.

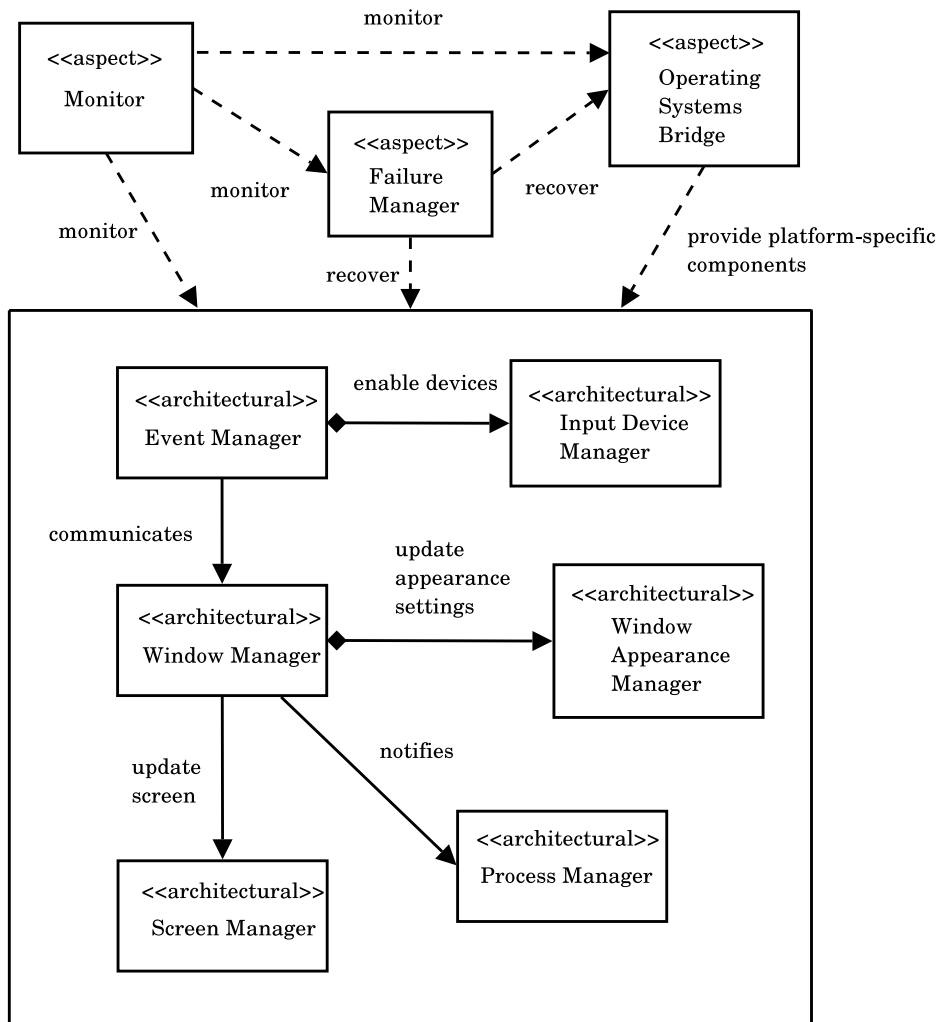


Figure 7.14: Transformed window manager software architecture

Chapter 8

Discussion and conclusions

8.1 Summary

This section provides a short summary of the entire thesis. The problem statement for was described in chapter 2. Several problems of existing software architecture evaluation methods were identified requirements for a new method have been defined. Chapter 3 have discusses background on DSMs, DMMs and their application to software architecture design. This overview of the background of DSMs and DMMs provided a context for an introduction to the Concern-Oriented Software Architecture Analysis Method, described in chapter 4. In that chapter, the phases of COSAAM are explained and the method is demonstrated with a window manager software architecture case study. The individual phases are described separately in the chapters 4, 5 and 6. These three chapters described a complete iteration of COSAAM. The remaining iterations of the COSAAM evaluation for the window manager were discussed in chapter 7. This chapter discusses our experiences and lessons learned during the the development and application of COSAAM and presents the main conclusions. Furthermore, we elaborate on opportunities for future research on the development and application of the method.

8.2 Discussion

The application of COSAAM to the window manager case study has delivered useful observations. The following sections describe some general prerequisites for an effective COSAAM evaluation. After these have been described the discussion assesses interesting findings identified during the evolution of the window manager software architecture. Finally, various opportunities for future research are discussed.

The application of COSAAM transformation rules requires access to problem- and solution domain knowledge. COSAAM can guide the evaluation team

during the evaluation process, but it cannot provide available domain knowledge that is necessary to design modules and define problems. Domain knowledge is important in software architecture design and remains important when using an evaluation method such as COSAAM [3]. The evaluation team can perform domain analysis and derive domain models or use existing knowledge, such as architectural patterns or design patterns.

COSAAM has not yet been applied in a actual project. As a result, guidelines for building effective evaluation teams have not been established. When COSAAM is validated by future case studies it should be possible to determine such guidelines. However, several general observations can be identified, based on the application of COSAAM to the window manager case study. Since COSAAM requires the perspectives of both concerns and the architecture, it is likely that a mixed group of stakeholders and software architects are necessary for an effective evaluation. The stakeholders continually clarify and refine the problem definition and the concerns that need to be addressed by the software architecture. The software architects, on the other hand, make technical decisions in order to transform the software architecture so it meets the concerns of stakeholders. Both the problem and solution perspectives are required for effective software architecture evaluation.

At the time of writing, COSAAM is not supported by a tool environment. A tool environment for COSAAM would be useful for the following reasons. First, without a tool environment, the manual updating of the DMM and DSM is time-consuming and error-prone. Tool support can fully automate the activities *Measure Scattering and Tangling*, *Initialize and Sequence Architecture DSM* and *Apply Transformation Rule*. In addition, the activities *Characterize Concerns and Modules Mapping* and *Select Transformation Rule* may be partially automated in a tool environment. Automation increases the productivity of the evaluation team and the quality of the evaluation. Chapter 6 provides a set of primitive DMM and DSM transformations with closure properties, which can help the development of tool support for COSAAM.

A second reason for tool support is manageability of the process. In all but the most trivial cases, the DMM and DSM can become quite large. A tool environment can provide options for filtering, compressing and sorting the matrices, which increases their readability and clarity. Tool support can also provide ways to reuse concerns or software architecture designs from a repository of previous evaluations.

A third benefit of tool support is that it creates a more responsive evaluation process. With a tool environment, the effect of a transformation rule is almost immediately visible. Version control of the DMM and DSM can provide an even greater flexibility during the evaluation. The evaluation team may branch and merge different versions of the DMM and DSM. In this way, the evaluation team can try different transformations and revert back to previous designs

and compare design alternatives during the transformation of the software architecture. The analysis of design alternatives is an implicit process in COSAAM, but is explicit in synthesis based software architecture design, which can be used as a compliment to COSAAM [42].

At the end of the evaluation and transformation of the window manager software architecture, scattering and tangling of concerns and modules has been eliminated. Especially in the last three iterations, scattering of concerns has been reduced considerably with the design of the *Operating Systems Bridge*, *Failure Manager*, and the *Monitor* aspects. During these three iterations, the architecture DSM showed an increase in the number of advice relationships. The increased amount and scope of advice relationships can cause problems, because aspects may interact at the same modules. This is shown in the DSM when there are multiple advice relationships in a row. If the window manager software architecture were to be implemented, the developers would have to specify exact ordering of execution of each aspect. The interaction of aspects is a complex problem and has been actively researched [7], [34], [30]. The evolution of the DMM and DSM show that an increase of the cohesion of modules, due to the lack of scattering and tangling, is contrasted with an increase in complexity caused by aspect interactions. Future research may investigate this trade-off between the cohesion of modules and the complexity of interacting aspects more thoroughly.

COSAAM demonstrates that the identification of crosscutting concerns at the architecture level is important for the development of software architectures. There are two reasons to support this statement. The first reason is based on our dependency analysis of the software architecture design and the transformation rules. Crosscutting concerns are identified early in the process but aspects are designed as late as possible, when the modules they cut across are cohesive and stable. The second reason is that identification of crosscutting concerns helps the transformation activity as a whole. We distinguish between candidate crosscutting concerns and inherent crosscutting concerns. This will help us in our analysis during the transformation activity. COSAAM shows the benefits of identifying crosscutting concerns at the software architecture level. However COSAAM does not provide heuristics to determine the ordering of the design of multiple aspects.

8.3 Conclusions

This section provides the main conclusions of the development of COSAAM and its application to the window manager case study.

- The application of COSAAM has eliminated scattering and tangling for the given set of concerns through the identification and design of the *Operating Systems Bridge*, *Failure Manager*, and the *Monitor* aspects for the window manager software architecture.

- COSAAM provides heuristics to identify crosscutting concerns early and defer the design of aspects. This separation between aspect identification and aspect design is important to minimize rework during architecture transformation
- The DMM and DSM show a possible trade-off between module cohesion and the complexity of module compositions from interacting aspects. High cohesion is an important quality for software architectures. However, the COSAAM application indicates that high cohesion can be contrasted with interacting aspects, which increase the complexity of the design.
- The ordering of the design of aspects is implicitly defined in COSAAM. Even though COSAAM uses a DSM to identify dependencies between existing modules, it is not clear how to identify future dependencies between aspects that have not yet been designed.
- The quality of a COSAAM evaluations depends on the quality of available domain knowledge.
- Tool support is required to validate COSAAM. Additionally, a more automated process provides greater flexibility during the evaluation.
- The analysis of design alternatives is an implicit process in COSAAM

8.4 Future Work

During transformation and software architecture design it is necessary to balance various quality factors, such as adaptability and performance. Design alternatives based on different quality factors have been represented by design spaces [42]. Design spaces can also be represented by a DMM that maps generic representations of modules to different design alternatives with respect to distinct quality factors. This extension would make the analysis of design alternatives more explicit during an evaluation.

In practice, software architecture design involves the definition of several architectural views to specify distinct qualities of a design [11] [23]. COSAAM investigates the mapping of concerns to elements of a specific architectural view. Further research may investigate the application of DMMs to the evaluation of multiple architectural views or mappings between architectural views. The DSM and DMM are general purpose tools and could be used in many situations.

Bibliography

- [1] Aspectj - <http://www.eclipse.org/aspectj>.
- [2] The ruby programming language.
- [3] M. Akşit. The 7 Cs for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering. *Turkish Journal of Electrical Engineering & Computer Sciences*, 12(2):61–95, 2004.
- [4] B. Tekinerdoğan & F. Scholten. Asaam-t: A tool environment for identifying architectural aspects. demo at aosd 2005 conference, chicago, 2005.
- [5] M. Ali Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. *aswec*, 00:309, 2004.
- [6] J. Bakker. Traceability of concerns. Master’s thesis, University of Twente, 2005.
- [7] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Workshop on Analysis of Aspect-Oriented Software*, ECOOP 2003, 2003.
- [8] B. Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, and T. Mens. A discussion of refactoring in research and practice, 2004.
- [9] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
- [10] T. Browning. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. volume 48, pages 292–306, New York, NY, USA, 2001. ACM Press.
- [11] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

- [12] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, January 2002.
- [13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
- [14] Sandkull B. Danilovic M. The use of dependence structure matrix and domain mapping matrix in managing uncertainty in mulitple project situations. *International Journal on Project Management*, 3:193–203, 2005.
- [15] E. Figueiredo et al. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. 2005.
- [16] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004. ISBN 0-32-121976-.
- [17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. FOW m 01:1 1.Ex.
- [18] J. Rumbaugh G. Booch and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., 1998.
- [19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [20] J. Bakker, B. Tekinerdoğan and M. Aksit. Characterization of Early Aspects Approaches. 2005.
- [21] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. SAAM: a method for analyzing the properties of software architectures. In *ICSE '94: Proceedings of the 16th national conference on Software engineering*, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [22] Jos Mara Conejero Klaas van den Berg and Juan Hernndez. Analysis of crosscutting across software development phases based on traceability. In *EA '06: Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 43–50, New York, NY, USA, 2006. ACM Press.
- [23] Philippe Kruchten. Architectural blueprints—The “4+1” view model of software architecture. *IEEE Software*, 12(6):42–50, nov 1995.
- [24] Lattix Inc. <http://www.lattix.com>.
- [25] Karl J. Lieberherr. Controlling the complexity of software designs. *icse*, 0:2–11, 2004.

- [26] Cristina Videira Lopes and Sushil Bajracharya. Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. In *T. Aspect-Oriented Software Development I*, pages 1–35, 2006.
- [27] Cristina Videira Lopes and Sushil Krishna Bajracharya. An Analysis of Modularity in Aspect Oriented Design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM Press.
- [28] R. C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Prentice-Hall, 2002.
- [29] Mattia Monga. On aspect-oriented approaches. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, sep 2004. German Informatics Society.
- [30] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. <http://trese.cs.utwente.nl/~nagyist/nagy2006.pdf>, IPA, May 2006. ISBN 90-365-2368-0.
- [31] Martin E. Nordberg and III. Aspect-oriented dependency inversion, 2001.
- [32] David L. Parnas. On the criteria to be used in decomposing systems into modules. pages 411–427, 2002.
- [33] R. Kazman and M. Klein and P. Clements. ATAM: Method for Architecture Evaluation, 2000.
- [34] Frans Sanen, Neil Loughran, Awais Rashid, Andronikos Nedos, Andrew Jackson, Siobhn Clarke, Eddy Truyen, and Wouter Joosen. Classifying and documenting aspect interactions. In *5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD 2006*, Bonn, Germany, 2006.
- [35] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 167–176, New York, NY, USA, 2005. ACM Press.
- [36] D. Sharman and A. Yassine. Characterizing Complex Product Architectures. *Systems Engineering Journal*, 7(1), 2004.
- [37] Donald Steward, Stephen Denker, and Tyson Browning. Planning Concurrency and Managing Iteration in Projects. *Center for Quality of Management Journal*, 8(2):55–62, 1999.
- [38] Donald V. Steward. The Design Structure System, A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, 1981.

- [39] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.
- [40] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The Structure and Value of Modularity in Software Design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, 2001.
- [41] B. Tekinerdoğan and M. Akşit. Providing automatic support for heuristic rules of methods, in object-oriented technology. In S. Demeyer and J. Bosch, editors, *ECOOP98 Workshop Reader*, pages 496–498. Springer Verlag, Jul 1998.
- [42] Bedir Tekinerdogan. *Synthesis-Based Software Architecture Design*. PhD thesis, University of Twente, Mar 2000.
- [43] Bedir Tekinerdoğan. ASAAM: Aspectual Software Architecture Analysis Method. In *Working IEEE/IFIP Conference on Software Architecture*, 2004.
- [44] Klaas van den Berg, Bedir Tekinerdogan, and Hoa Nguyen. Analysis of crosscutting in model transformations. In J. Oldevik J. Aagedal, T. Neple, editor, *ECMDA-TW Traceability Workshop Proceedings 2006*, number A219 in SINTEF Report, pages 51–64, 2006. ISBN 82-14-04030.
- [45] Gerald M. Weinberg. *Quality Software Management (Vol. 2): First-Order Measurement*. Dorset House Publishing Co., Inc., New York, NY, USA, 1993.
- [46] A. Yassine. An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method. 2002.
- [47] C. Zhang and H. Jacobsen. Re-factoring middleware with aspects, 2003.

Appendix A

DSM clustering algorithm source code

The following pages contain the source code for the clustering algorithm used during the case study. Since a DSM is graph, we have used RGL, the ruby graph library to model DSMs.

File: DSM.rb

```
require 'rgl/adjacency'
require 'Cluster'

class DSM < RGL::AdjacencyGraph

  attr_accessor :clusters

  def initialize
    super
    @clusters = Array.new()
    @weights = Hash.new()
    self.to_undirected
  end

  def cell(i,j)
    if has_edge?(i,j) or has_edge?(j,i)
      return 1
    else
      return 0
    end
  end

  def getClustersForItem(i)
    clusters = Array.new()
    @clusters.each{ |c|
      if c.include?(i)
        clusters << c
      end
    }
  end
end
```

```

    }
    return clusters
end

def to_s
  s = ""
  @clusters.each{ |c|
    s += "Cluster #{c.id}:"
    c.each{ |i|
      s += " #{i}"
    }
    s += "\n"
  }
  return s
end

def get(id)
  element = self.detect{ |e| e.id == id }
  return element
end

end

```

File: Cluster.rb

```
class Cluster < Array

  attr_accessor :id

  def initialize()
    super()
  end

  def <<(e)
    if !self.include?(e)
      self.push e
    end
  end

  def id
    @id
  end

  def to_s
    string = "Cluster:\n"
    self.each{ |v| string += " #{v.to_s}\n" }
    return string
  end

end
```

File: Scenario.rb

```
class Scenario

  attr_accessor :id, :description

  def initialize(id,description)
    @id          = id
    @description  = description
  end

  def id
    @id
  end

  def description
    @description
  end

  def to_s
    return "#{@id}"
  end

end
```

File: DegreeBasedClusteringAlgorithm.rb

```
require 'DSM'
require 'Cluster'

class DegreeBasedClusteringAlgorithm

  attr_accessor :dsm, :queue, :passed

  def initialize(dsm)
    @dsm      = dsm
    @queue     = Array.new(@dsm.vertices)
    @passed    = Array.new()
  end

  def sortOnDegree
    @queue = @queue.sort{ |i,j| @dsm.out_degree(j) <=> @dsm.out_degree(i) }
  end

  def queue
    @queue
  end

  def clusterElement(v)
    c = Cluster.new()
    c << v
    @dsm.each_adjacent(v) { |w|
      c << w
      @passed << w
    }
    @dsm.clusters << c
  end

  def run
    sortOnDegree()
    @passed = Array.new()
    @queue.each{ |v|
      if @passed.include?(v)
        next
      end
      clusterElement(v)
      @passed << v
    }
  end
end
```

end

File: TestDegreeBasedClusteringAlgorithm.rb

```
require 'test/unit'
require 'DegreeBasedClusteringAlgorithm'

require 'DSM'
require 'Cluster'
require 'Scenario'

class TestDegreeBasedClusteringAlgorithm < Test::Unit::TestCase

  attr_accessor :algo, :dsm, :s1, :s2, :s3, :s4, :s5, :s6, :s7, :s8, :s9, :s10,
    :s11, :s12, :s13, :s14, :s15, :s16, :s17, :s18, :s19, :s20, :cluster

  def setup
    @s1 = Scenario.new("S1","Start multiple processes at the same time")
    @s2 = Scenario.new("S2","change color of widgets in a window")
    @s3 = Scenario.new("S3","Close all open windows")
    @s4 = Scenario.new("S4","Change screen resolution")
    @s5 = Scenario.new("S5","Enter a command to start an application process")
    @s6 = Scenario.new("S6","Move the main window")
    @s7 = Scenario.new("S7","Screen saver is activated")
    @s8 = Scenario.new("S8","Resize a window")
    @s9 = Scenario.new("S9","Terminate a process")
    @s10 = Scenario.new("S10","Interrupt a process")
    @s11 = Scenario.new("S11","Change look-and-feel style at run time")
    @s12 = Scenario.new("S12","Add voice control")
    @s13 = Scenario.new("S13","A failure occurs and the system shuts down")
    @s14 = Scenario.new("S14","Provide dual display screen")
    @s15 = Scenario.new("S15","Use multiple desktops")
    @s16 = Scenario.new("S16","Monitor activities of the user")
    @s17 = Scenario.new("S17","Provide touch screen and light pen as input")
    @s18 = Scenario.new("S18","A memory overflow due to too many opened windows")
    @s19 = Scenario.new("S19","Port system to command based operating system")
    @s20 = Scenario.new("S20","Minimize windows after idle time")
  end

  def simpleDSM
    @dsm = DSM.new()
    @dsm.to_undirected

    @s1 = Scenario.new("S1","Start multiple processes at the same time")
    @dsm.add_vertex(s1)
    @s2 = Scenario.new("S2","change color of widgets in a window")
```



```

@dsm.add_vertex(s2)
@s3 = Scenario.new("S3","Close all open windows")
@dsm.add_vertex(s3)
@s4 = Scenario.new("S4","Change screen resolution")
@dsm.add_vertex(s4)
@s5 = Scenario.new("S5","Add voice control")
@dsm.add_vertex(s5)
@s6 = Scenario.new("S6","Change color of widget in a window")
@dsm.add_vertex(s6)

@dsm.add_edge @s1, @s2
@dsm.add_edge @s1, @s3
@dsm.add_edge @s1, @s4
@dsm.add_edge @s1, @s5

@dsm.add_edge @s2, @s6

@dsm.add_edge @s3, @s4
@dsm.add_edge @s3, @s5

@dsm.add_edge @s4, @s5

@algo = DegreeBasedClusteringAlgorithm.new(@dsm)

end

def teardown
  @dsm = nil
  @algo = nil
end

def testDSMNotNil
  simpleDSM()
  assert_not_nil(@dsm)
end

def testAlgoNotNil
  simpleDSM()
  assert_not_nil(@algo)
end

def testAlgoDSMNotNil
  simpleDSM()
  assert_not_nil(@algo.dsm)
end

```

```

def testSortOnDegree
  simpleDSM()
  @algo.sortOnDegree()

  max = 0
  @algo.queue.each { |s|
    if @dsm.out_degree(s) >= max
      max = @dsm.out_degree(s)
    end
  }
  assert_equal(max,@dsm.out_degree(@algo.queue.first))
  assert_equal(@dsm.size,@algo.queue.size)
end

def testGet
  simpleDSM()
  scenario = @dsm.get("S1")
  assert_equal(@s1,scenario)
end

def testDegree(n,v)
  assert_equal(n,v)
end

def clusterDSMWithIslandNode
  @dsm = DSM.new()

  @s1 = Scenario.new("S1","Start multiple processes at the same time")
  @dsm.add_vertex @s1

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)
  @algo.run()
end

def clusterDSMWithSpikeNode
  @dsm = DSM.new()

  @s13 = Scenario.new("S13","A failure occurs and the system shuts down")
  @dsm.add_vertex @s13
  @s18 = Scenario.new("S18","A memory overflow due to too many opened windows")
  @dsm.add_vertex @s18
  @dsm.add_edge @s13, @s18

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)
  @algo.run()
end

```

```

def testIslandNodeClustering
  @dsm = DSM.new()

  @s1 = Scenario.new("S19", "Port system to command-based operating system")
  @dsm.add_vertex @s1

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)
  @algo.run()

  assert_equal(1, @dsm.clusters.size) # One cluster is created
  assert_equal(true, @dsm.clusters.first.include?(@s1)) # S1 is in the cluster
end

def testSpikeNodeClustering
  @dsm = DSM.new()

  s13 = Scenario.new("S13", "A failure occurs and the system shuts down")
  @dsm.add_vertex s13
  s18 = Scenario.new("S18", "A memory overflow due to too many opened windows")
  @dsm.add_vertex s18
  @dsm.add_edge s13, s18

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)
  @algo.run()

  assert_equal(1, @dsm.clusters.size) # One cluster is created
  assert_equal(2, @dsm.clusters.first.size) # Two scenarios in the cluster
end

def testMultipleSpikeNodeClusterings
  @dsm = DSM.new()

  s13 = Scenario.new("S13", "A failure occurs and the system shuts down")
  @dsm.add_vertex s13
  s18 = Scenario.new("S18", "A memory overflow due to too many opened windows")
  @dsm.add_vertex s18
  @dsm.add_edge s13, s18

  s6 = Scenario.new("S6", "Move the main window")
  @dsm.add_vertex s6
  s8 = Scenario.new("S8", "Resize a window")
  @dsm.add_vertex s8
  @dsm.add_edge s6, s8

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)

```

```

@algo.run()

assert_equal(2,@dsm.clusters.size)      # Two cluster are created
assert_equal(2,@dsm.clusters.first.size) # Two scenarios in the first cluster
assert_equal(2,@dsm.clusters[1].size)    # Two scenarios in the second cluster
end

# Queue

def testElementQueueInit
  @dsm = DSM.new()

  @s1 = Scenario.new("S1","Start multiple processes at the same time")
  @dsm.add_vertex @s1

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)

  assert_equal(@dsm.vertices,@algo.queue)      # Queue is equal to the vertice set
end

def testElementQueueAfterIslandNodeClustering
  clusterDSMWithIslandNode()

  assert_equal(true,@algo.passed.include?(@s1)) # Scenario is visited from the queue
end

def testElementQueueAfterSpikeNodeClustering
  clusterDSMWithSpikeNode()

  assert_equal(true,@algo.passed.include?(@s13)) # Scenarios is marked as visited after
  assert_equal(true,@algo.passed.include?(@s18))
end

def testTriadClustering
  @dsm = DSM.new()

  @dsm.add_vertex @s7
  @dsm.add_vertex @s16
  @dsm.add_vertex @s20

  @dsm.add_edge @s7, @s20
  @dsm.add_edge @s7, @s16
  @dsm.add_edge @s16, @s20

  @algo = DegreeBasedClusteringAlgorithm.new(@dsm)
  @algo.run()

```

```

    assert_equal(1,@dsm.clusters.size)
    assert_equal(3,@dsm.clusters.first.size)
end

```

```

def testOverlappingClusters

```

```

    @dsm = DSM.new()

```

```

    @dsm.add_vertex @s17
    @dsm.add_vertex @s12
    @dsm.add_vertex @s5
    @dsm.add_vertex @s10
    @dsm.add_vertex @s1
    @dsm.add_vertex @s8
    @dsm.add_vertex @s6
    @dsm.add_vertex @s9
    @dsm.add_vertex @s3

```

```

    @dsm.add_edge @s6, @s8
    @dsm.add_edge @s8, @s12
    @dsm.add_edge @s12, @s17
    @dsm.add_edge @s12, @s5
    @dsm.add_edge @s12, @s10
    @dsm.add_edge @s10, @s1
    @dsm.add_edge @s5, @s1
    @dsm.add_edge @s1, @s9
    @dsm.add_edge @s9, @s3

```

```

    @dsm.add_vertex @s13
    @dsm.add_vertex @s18

```

```

    @dsm.add_edge @s13, @s18

```

```

    @dsm.add_vertex @s16
    @dsm.add_vertex @s20
    @dsm.add_vertex @s7
    @dsm.add_vertex @s4
    @dsm.add_vertex @s11
    @dsm.add_vertex @s15
    @dsm.add_vertex @s14

```

```

    @dsm.add_edge @s16, @s20
    @dsm.add_edge @s7, @s20
    @dsm.add_edge @s7, @s16
    @dsm.add_edge @s7, @s4
    @dsm.add_edge @s4, @s14

```

```

@dsm.add_edge @s14, @s15
@dsm.add_edge @s4, @s15
@dsm.add_edge @s4, @s11
@dsm.add_edge @s15, @s11
@dsm.add_edge @s2, @s11

@dsm.add_vertex @s19

@algo = DegreeBasedClusteringAlgorithm.new(@dsm)
@algo.run()

# Cluster 1
c = @dsm.getClustersForItem(@s12).first
assert_equal(true,c.include?(@s17))
assert_equal(true,c.include?(@s5))
assert_equal(true,c.include?(@s10))
assert_equal(true,c.include?(@s8))

# Cluster 2
c = @dsm.getClustersForItem(@s6).first
assert_equal(true,c.include?(@s8))

# Cluster 3
c = @dsm.getClustersForItem(@s1).first
assert_equal(true,c.include?(@s5))
assert_equal(true,c.include?(@s10))
assert_equal(true,c.include?(@s9))

# Cluster 4
c = @dsm.getClustersForItem(@s3).first
assert_equal(true,c.include?(@s9))

# Cluster 5
c = @dsm.getClustersForItem(@s19).first
assert_equal(1,c.size)

# Cluster 6
c = @dsm.getClustersForItem(@s2).first
assert_equal(true,c.include?(@s11))

# Cluster 7
c = @dsm.getClustersForItem(@s13).first
assert_equal(true,c.include?(@s18))

# Cluster 8
c = @dsm.getClustersForItem(@s4).first

```

```

    assert_equal(true,c.include?(@s14))
    assert_equal(true,c.include?(@s15))
    assert_equal(true,c.include?(@s11))
    assert_equal(true,c.include?(@s7))

    # Cluster 9
    c = @dsm.getClustersForItem(@s20).first
    assert_equal(true,c.include?(@s16))
    assert_equal(true,c.include?(@s7))

    # Test overlapping elements
    assert_equal(2,@dsm.getClustersForItem(@s8).size)
    assert_equal(2,@dsm.getClustersForItem(@s10).size)
    assert_equal(2,@dsm.getClustersForItem(@s5).size)
    assert_equal(2,@dsm.getClustersForItem(@s9).size)
  end
end

```