Evaluating Complexity of Aspect-Oriented Software Development Comparing to Use Case Driven Software Development

By KIATSOONGSONG Weerayut

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu

September, 2011

Evaluating Complexity of Aspect-Oriented Software Development Comparing to Use Case Driven Software Development

By KIATSOONGSONG Weerayut (0910204)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu

and approved by
Professor Koichiro Ochimizu
Associate Professor Masato Suzuki
Associate Professor Toshiaki Aoki

August, 2011 (Submitted)

Abstract

Use case driven software development (UDSD) is an approach that is mostly used in the software engineering industry to develop software system by using use cases. But when each use case is realized, the components fulfilling a certain use case are spread through the system and cannot be kept separate from other use cases' components. This situation is called crosscutting concerns. As a result, it reduces the maintainability of the system. Aspect-oriented software development (AOSD) with use cases proposed by Ivar Jacobson is said to be an approach which helps increase maintainability and reduce the effect of crosscutting concerns of the system implemented by UDSD by using aspect and use-case slice to encapsulate the modules and part of modules that are specific to a certain use case. However, there still has no evidence to prove efficiency of AOSD over UDSD.

Our research proposed the way to evaluate how much AOSD helps increase the maintainability and how much AOSD helps reduce the effect of crosscutting of the UDSD system. First, we instantiated concern-oriented meta-model proposed by Figueiredo, E. et al. as a base for comparing UDSD and AOSD system. Second, we proposed one metrics suite called change impact metrics suite to evaluate how the change affects the system after the change occurs. Since, the maintainability refers directly to the change; this metrics suite can refer to the maintainability of the system. This metrics suite was defined based on the number of components and relationships in each artifact created from requirements phase to design phase. Third, we applied one metrics suite called scattering, tangling, and crosscutting metrics suite proposed by Conejero J. et al. to evaluate how much the separation of concerns in the system. Then, we used the ATM system which is introduced in "Designing Concurrent, Distributed, and Real-Time Applications with UML" as our case study and implemented ATM system for both UDSD and AOSD approach from requirements phase to design phase. Finally, we measured our metrics from both systems.

The results of our empirical study show that the AOSD system is more maintainable and has less effect of crosscutting concerns than the UDSD system. However, there is only small difference between measures of all the metrics. This is because in the real world some classes contain not only the parts (methods and attributes) that fulfill different use cases and are not related to each other, but also the parts that are used by many use cases, which we called common parts. AOSD provides non-use-case-specific slice to contain these common parts. As a result, use case still has the dependency to the base class in common parts and some classes still has relationships to the base classes, so when the change occurs, it can also affect these common parts. Therefore, the efficiency of AOSD is hindered by the use of non-use-case-specific slices. However, if we remove the use of non-use-case-specific slices out of the system, it reduces the reusability of the system and increases the system size because of the common parts. Consequently, we have to consider the trade-off between separation of concerns and reusability.

Acknowledgement

I would like to express my gratitude to those who gave me the opportunity to complete this dissertation. This dissertation could not have been written without Professor Koichiro Ochimizu who not only served as my supervisor but also encouraged and challenged me throughout my academic program, Associate Professor Masato Suzuki and Associate Professor Toshiaki Aoki who took time from their busy schedules to serve me on my dissertation committee and gave me a lot of constructive criticisms, and Ms. CAMARGO CRUZ Ana Erika who gave me a lot of useful guidances and opinions to continue my research. Moreover, I would like to express my thanks to every member of our lab for the encouragement and support on my work and learning experience. Last, I would like to thank JAIST and Japan for giving me such good experiences.

Contents

1 Introduction **1**

2 Background **4**
2.1 Use Case Driven Software Development 4
2.1.1 Use Case Driven Software Development Process and Artifacts . . . 4
2.1.2 Use Case Driven Software Development Pitfalls 6
2.2 Aspect-Oriented Software Development 7
2.2.1 Aspect and Use-case slice 7
2.2.2 Aspect-Oriented Software Development Process and Artifacts . . . 11
2.3 Metrics and Measurement 12

3 Related Works **15**
3.1 Separation of Concerns Metrics 15
3.1.1 Sant'Anna C.'s Metrics 15
3.1.2 Conejero J.'s Metrics 16

4 Our Approach **18**
4.1 How to Compare UDSD and AOSD 18
4.2 Our Metrics Suites for Evaluating UDSD and AOSD 20
4.2.1 Change Impact Metrics Suite 20
4.2.2 Scattering, Tangling, and Crosscutting Metrics Suite 23

5 Empirical Study and Results **30**
5.1 Case Study: ATM System 30
5.1.1 Problem Description 30
5.1.2 Use Case Model 31
5.1.3 Analysis Model and Design Model 32
5.1.4 System after Change 38
5.2 Results of Measurement 39
5.2.1 Measures of Change Impact Metrics 39
5.2.2 Measures of Scattering, Tangling and Crosscutting Metrics 40
5.3 Statistical Analysis for Our Results Using T-Test 41
5.3.1 T-Test Definition and Procedure 42
5.3.2 T-Test Calculation for Our Metrics Results 44

5.4 Discussion 49
5.4.1 Discussion for Change Impact Metrics 49
5.4.2 Discussion for Scattering, Tangling, and Crosscutting Metrics . . . 50
5.5 Effect of AOSD Characteristic on Our Results 51
5.5.1 The Ideal Case and Practical Case for Crosscutting Concerns 51
5.5.2 The Use of Non-Use-Case-Specific Slice 54
5.5.3 AOSD System without Non-Use-Case-Specific Slice 56

6 Conclusion and Future Works **62**

A Use Case Description of the Case Study: ATM Sytem **66**

List of Figures

2.1 Tangling and Scattering Example [2] 7
2.2 Use-case Slice Example [2] 8
2.3 Use-Case Realization Example: An Interaction Diagram for the *Reserve Room* Use-Case Realization [2] 9
2.4 Use-Case Slice Example: *Reserve Room* Use-Case Slice [2] 10
2.5 Another Use-Case Realization Example: An Interaction Diagram for the *Check In Customer* Use-Case Realization [2] 10
2.6 Non-Use-Case-Specific Slice Example [2] 11

4.1 Abstract Meta-Model of Aspect-Oriented Systems [16] 19
4.2 System after Change 23
4.3 Abstract Meta-Model of the Crosscutting Pattern [15] 24

5.1 ATM System Use Case Diagram 32
5.2 Example of UDSD Analysis Model - Class Diagram for *Validate PIN* Use Case 33
5.3 Example of UDSD Analysis Model - Collaboration Diagram for *Validate PIN* Use Case Typical Flow 34
5.4 Example of AOSD Analysis Model - Use-Case Slice for *Validate PIN* Use Case 35
5.5 Example of AOSD Analysis Model - Use-Case Slice with Non-Use-Case-Specific Slice for *Validate PIN* Use Case 36
5.6 Example of AOSD Analysis Model - Collaboration Diagram for *Validate PIN* Use Case Typical Flow 37
5.7 ATM System Use Case Diagram after Applying Change 38
5.8 Explanation of Lower Change Impact in AOSD Comparing to UDSD . . . 49
5.9 The Ideal Case for Crosscutting Concerns 52
5.10 Use-Case Slice and Aspect for Ideal Crosscutting Concerns 53
5.11 The Practical Case for Crosscutting Concerns 54
5.12 *Validate PIN* Use-Case Slice Extending the Four Non-Use-Case-Specific Slices 55

List of Tables

2.1 UDSD process and artifacts 6
2.2 AOSD process and artifacts 12

4.1 Meta-Model Instantiation for UDSD and AOSD Systems 20
4.2 Component and Relationship of System's Diagrams 21
4.3 Crosscutting Meta-Model Instantiation for UDSD and AOSD Systems . . . 25
4.4 Example of Dependency Matrix 25
4.5 Example of Tangling Matrix 26
4.6 Example of Crosscutting Product Matrix 26
4.7 Example of Crosscutting Matrix 27
4.8 Summary of the Scattering, Tangling, and Crosscutting Metrics 29

5.1 Measures of Change Impact Metric 39
5.2 Measures of Scattering and Crosscutting Metric of UDSD System 40
5.3 Measures of Scattering and Crosscutting Metric of AOSD System 41
5.4 Measures of Tangling Metric 41
5.5 Difference of Measures between UDSD and AOSD 42
5.6 T-Test Formulas 43
5.7 T-Test Calculation for Degree of Change Impact I Measures 44
5.8 T-Test Calculation for Scattering Metric Measures at Analysis 45
5.9 T-Test Calculation for Scattering Metric Measures at Analysis 46
5.10 T-Test Calculation for Tangling Metric Measures at Analysis 47
5.11 T-Test Calculation for Tangling Metric Measures at Design 47
5.12 T-Test Calculation for Crosscutting Metric Measures at Analysis 48
5.13 T-Test Calculation for Crosscutting Metric Measures at Design 48
5.14 Results of ATM System Implemented by AOSD without NUCS 56
5.15 T-Test Calculation for Degree of Change Impact I Measures (for UDSD and AOSD without NUCS) 57
5.16 T-Test Calculation for Scattering Metric Measures at Analysis (for UDSD and AOSD without NUCS) 58
5.17 T-Test Calculation for Scattering Metric Measures at Design (for UDSD and AOSD without NUCS) 58
5.18 T-Test Calculation for Tangling Metric Measures at Analysis (for UDSD and AOSD without NUCS) 59

5.19 T-Test Calculation for Tangling Metric Measures at Design (for UDSD and AOSD without NUCS) 59
5.20 T-Test Calculation for Crosscutting Metric Measures at Analysis (for UDSD and AOSD without NUCS) 60
5.21 T-Test Calculation for Crosscutting Metric Measures at Design (for UDSD and AOSD without NUCS) 61

A.1 *Validate PIN* Use Case Description 67
A.2 *Withdraw Funds* Use Case Description 68
A.3 *Query Account* Use Case Description 69
A.4 *Transfer Funds* Use Case Description 70
A.5 *Cancel Transaction* Use Case Description 71
A.6 *Borrow Money* Use Case Description (Addition According to the Change) . 72

Chapter 1

Introduction

Nowadays, in the software industry, use case driven software development (UDSD) has broadly been used in order to increase understandability and reusability of the software systems. On the basis of Object-Oriented Approach (OOA), use case driven software development complements OOA by providing the unified software development process. In this process, software developers use use-case model to capture the requirements from the customer's needs and represent requirements in a suitable way in order to facilitate the communication amongst software stakeholders (users, customers and developers). Moreover, the use cases drive through the whole development process. In other words, use cases are used as a base from requirement gathering phase through the whole software life cycle [1].

However, after capturing use cases from requirements, the use cases cannot literally be kept separate throughout the whole development process. During the transition from requirement gathering phase to analysis-design phase, or in use case driven software development, from use case specification to use case realization, there occur two problems; scattering and tangling. Scattering is a situation that the codes that realize a particular use case are spread across multiple components of the system. And tangling is a situation that each component in the system contains the implementation to satisfy different use cases. As a consequence, use cases cut across the system and then use cases are not kept separate from each other [2].

A concern is some part of the problem that we want to treat as a single conceptual unit [4]. Sometimes, a concern affects more than one component and a component contains parts of multiple concerns. These situations are called scattering and tangling respectively. These kind of concerns are called crosscutting concern. The consequences caused by these concerns are reduced comprehensibility, ease of evolution and reusability of software artifacts. Accordingly, these concerns should be separated.

Over the past decades, aspect-oriented programming (AOP) has been used in order to modularize crosscutting concerns at the implementation phase [3]. But the earlier crosscutting concerns are modularized, the more stable the software system structure is. Consequently, Ivar Jacobson proposed aspect-oriented software development (AOSD) with use cases to complement the concept of aspect-oriented programming. Aspect-oriented software development is a holistic approach to developing software systems with aspects

from requirements, to analysis and design, to implementation and test. Its process was defined based on the concept of UDSD and is said to be the approach that helps reduce the effect of scattering and tangling of UDSD. As a result, the software systems built by AOSD is said to have more maintainability than those built by UDSD.

Although AOSD has a possibility to improve such problems in UDSD, but there are still no evidence yet. Intuitively, AOSD may be more complex than UDSD by just counting the number of documents or line of codes (LOC) because AOSD adds more components to the system. But only physical measurement is not enough to conclude which approach is a better one. Accordingly, we have to define proper metrics to evaluate complexity of software systems implemented by both approaches. And then, we can compare those two systems implemented in different approaches according to the measures of the defined metrics.

This paper reports the results of our research on the evaluation of AOSD complexity in the comparison to UDSD complexity to see how much AOSD can help improve maintainability in UDSD, and how much AOSD can help reduce the effect of crosscutting concerns. In order to evaluate these two approaches, we proposed one metric suite called the change impact metrics suite and we applied metrics suite proposed by Conejero J. et al. [15] to our research. This metrics suite consists of scattering, tangling, and crosscutting metrics.

In order to measure the maintainability of the AOSD and UDSD system, the change impact metrics suite are defined to measure how much the system is affected by the change when the change requirements is added to the system. Since, maintainability means the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [17]. Therefore, measuring change impact is directly related to the maintainability of the system. The change impact metrics suite are calculated based on the number of components and relationships in the artifacts created at design level of both UDSD and AOSD systems.

In order to measure the effect of crosscutting concerns on the UDSD system and AOSD system, we applied scattering, tangling, and crosscutting metrics suite proposed by Conejero J. et al. Since, their metrics suite was well-defined and they can straightforwardly extract the effect of crosscutting concerns of the system. In their research, they used their metrics suite to measure the crosscutting concerns at requirements phase by using use case description as materials. However, in our research, we apply this metrics suite to measure the crosscutting concerns of the system at analysis and design phase by using class diagrams as materials in UDSD system and use-case slice as materials in AOSD system.

Metrics that are used in our research are product metrics. Since, the process of UDSD and the process of AOSD proposed by Ivar Jacobson are similar because AOSD process has been developed from UDSD process. Consequently, we would rather compare the products of these two approaches than their processes. Moreover, our metric suites are objective because our metrics have been defined in mathematical terms, so the observers can apply these metrics as many times with the same results. Lastly, our metrics are computed metrics because we computed from number of components and their relationships.

This thesis is divided into the following Chapters: In Chapter 2: Background, we

present a brief explanation about background knowledge needed to carry out our research study. In Chapter 3: Related Works, we present some of the research work has been done on the metrics that measure crosscutting concerns. In Chapter 4: Our Approach, we present the approach that we use in our research in order to achieve our objectives. First, we describe about how to compare these two approaches. Then, we explain about metrics suites used in our research; change impact metrics suite and crosscutting concerns metrics suite. In Chapter 5: Empirical Study and Results, we presents the case study and the results to validate our metrics. In Chapter 6: Conclusion and Future Works, we draw out our conclusions and describe out plans for future works.

Chapter 2

Background

This chapter presents a brief explanation about background knowledge needed to carry out our research study. First, we introduce the concept of use case driven software development to understand the process and the artifacts of this approach. Second, we provide the concept of aspect-oriented software development to understand the difference between this approach and use case driven software development. Third, we explain about the basic knowledge of metrics and measurement to apply to the evaluation of these two approaches' goodness.

2.1 Use Case Driven Software Development

A software system is brought into existence to serve its users. Therefore, to build a successful system we must know what its prospective users want and need. The term user refers not only to human users but to other systems that interact with the system being developed. An interaction of this sort is a use case. A use case is a piece of functionality in the system that gives a user a result of value. All use cases together make up the use case model which describes the complete functionality of the system.

Use Case Driven Software Development (UDSD) is an approach to develop software system by using use cases. Using use cases has two major merits. First, use cases offer a systematic and intuitive means of capturing functional requirements and they can be used as a means to communicate amongst software stakeholders (users, customers, and developers). Second, they drive the whole development process since most activities such as analysis, design, and test are performed starting from use cases. This leads to the increasing of system's understandability and maintainability because we can easily realize the system requirements and easily organize them during the development process [1].

2.1.1 Use Case Driven Software Development Process and Artifacts

In use case driven software development, we can divide the development phases into five phases and in each phase, there are artifacts created as the intermediary products of

the system. With the usage of UML, we can create the artifacts in the systematic way. The five phases of UDSD process are as follows;

- Requirements

  In this phase, developers capture requirements from customer's needs using use case model. A use-case model is a model of a system containing actors and use cases and their relationships. The major artifacts of requirements phase are a use case diagram and a use case description of each use case.

- Analysis

  In analysis, developers analyze the requirements as described in requirements capture by refining and structuring them. The purpose of doing this is to achieve a description of the requirements that is easy to maintain and that helps developers give structure to the whole system. Based on each use case, developers create use case realization which consists of static and dynamic behavior of the use case. The major artifacts of this phase are class diagram and collaboration diagram from each use case.

- Design

  In design, developers shape the system and find its form that lives up to all requirements including nonfunctional requirements and platform-specific constraints. The artifacts of this phase are based on analysis's artifacts. The major artifacts of this phase are the same as analysis; class diagrams and collaboration diagrams, but add more implementation viewpoint to the system.

- Implementation

  In implementation, developers start with the result from design and implement the system in terms of components, that is, source code, binaries, and so on. The major artifacts of this phase are source codes.

- Test

  In the test workflow, developers verify the result from implementation. The major artifacts of this phase are test cases and test results.

All the phases and artifacts of UDSD are shown in Table 2.1. In our research study, we focus on the phases before implementation, those are, requirements, analysis, and design phase. Since, before implementation, we design the system that can be seamlessly transformed to implementation. Consequently, the design artifacts can represent the architecture baseline of the whole system. Moreover, as one of the factors of decision making on which approach to be applied between UDSD and aspect-oriented software development, the earlier phases are better choice than implementation phase.

Table 2.1: UDSD process and artifacts

Phase	Artifacts
Requirements	use case diagram and use case description
Analysis	class diagram and collaboration diagram
Design	class diagram and collaboration diagram
Implementation	source code
Test	test case and test results

2.1.2 Use Case Driven Software Development Pitfalls

Although, UDSD is said to be a good approach for software development, there occur problems. After developers define the use cases in requirements phase, they have to transform these use cases into the developers' viewpoint in the analysis and design phase. They define the components and their relationships according to each use case. This activity can be called use case realization. In this activity, use cases should have been separated from each other, but on the other hand, there are two problems occur. These problems are:

- Tangling

  When all use cases are realized, all components and their relationships are defined. There are some components that, instead of single-mindedly fulfilling a particular use case, contain the implementation to satisfy different use cases. This situation is called tangling because parts of use cases tangle together. This hinders understandability and makes the learning curve stepper for developers.

- Scattering

  When all use cases are realized, there are codes that realize a particular use case are spread across multiple components. This situation is called scattering, because parts of use case are scattered throughout the system. From this situation, if the requirements about that use case change, or if the design of that use case changes, developers must update many components.

The example of these two problems is shown in Figure 2.1. In this figure, the hotel management system is used to describe the tangling and scattering. Hotel management system has three use cases; reserve room, check in customer, and check out customer. After use case realization, there are seven components in the system; customer screen, staff screen, reserve room, check in, check out, reservation, and room. For reserve room use case, there are four components spread across the system. Consequently, scattering occurs in this system. Moreover, for room component, there are three parts in order to fulfill the three use cases. This is called tangling [2].

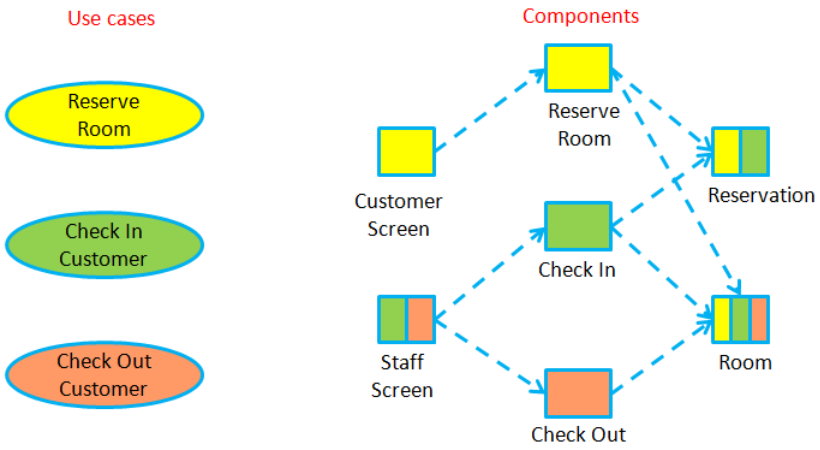

Figure 2.1: Tangling and Scattering Example [2]

2.2 Aspect-Oriented Software Development

A concern is some part of the problem that we want to treat as a single conceptual unit [4]. Sometimes, a concern affects more than one component and a component contains parts of multiple concerns. These situations are called scattering and tangling respectively. These kind of concerns are called crosscutting concern. The consequences caused by these concerns are reduced comprehensibility, ease of evolution and reusability of software artifacts. Accordingly, these concerns should be separated.

Aspect-oriented programming (AOP) is a programming paradigm that gives developers the means to separate code that implements crosscutting concerns and modularize it into aspects. Aspect-orientation provides the mechanism to compose crosscutting behaviors into the desired operations and classes during compile time and even during execution [3].

However, in order to progress beyond AOP, we need a holistic approach to develop software systems with aspects from requirements, to analysis and design, to implementation and test. This is aspect-oriented software development (AOSD). Ivar Jacobson and Pan Wei NG applied use case concept to AOSD. They use the use cases as a representation of concerns. Moreover, they proposed the concept of use-case slice that is used to separate the use cases from each other. The detail of use-case slice will be described in Section 2.2.1.

2.2.1 Aspect and Use-case slice

AOP introduced new constructs in order to separate and modularize concerns. These constructs are:

- Intertype declarations

  Intertype declarations allow developers to compose new features (attributes, operations, and relationships) into existing classes.

- Advices

  Advices provide the means to extend existing operations at extension points designated by pointcuts in AOP.

- Aspects

  Aspects are a kind of building block used to organize intertype declarations and advices.

With the concept of aspects, we can separate some features of classes into separate building blocks and separate extension features from the base features. Similar to this concept, Ivar Jacobson and Pan Wei NG proposed the concept of use-case slice to preserve the separation of concerns though the use case realization and implementation. Use-case slice is a modularity unit that collates the specifics of a use case during use case realization. Each use-case slice collates parts of classes, operations and so forth, that are specific to a use case in a model. The task of composing these parts is left to some composition mechanisms provided by AOP.

The example of use-case slice is shown in Figure 2.2. We use the same example in section 2.1.2, the hotel management system, to illustrate the difference between UDSD and AOSD. In this figure 2.2, the horizontal axis shows the element structure that identifies the classes in the system. The vertical axis shows the use case structure. It identifies the use cases being realized, each with a different shade. Each horizontal row depicts a use-case slice containing the extensions of classes needed to realize the use case for that row. Thus, we have the *ReserveRoom* use-case slice, the *CheckInCustomer* use-case slice, and the *CheckOutCustomer* use-case slice.

Each use-case slice contains partial class definitions specific to the use case realization. If we want complete class definitions, all we need to do is merge all the use-case slices.

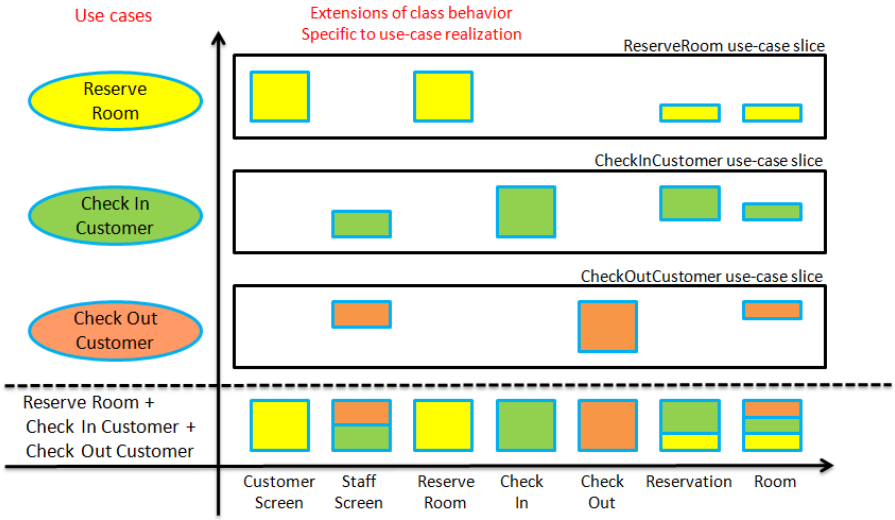

Figure 2.2: Use-case Slice Example [2]

Let us look at the use-case slice in more detail. Use-case slice contains the following:

1. A collaboration that describes the realization of the use case.
2. Classes specific to the use-case realization.
3. Extensions of existing classes specific to the use-case realization.

For example, the *Reserve Room* use case has simple event flows as shown in Figure 2.3. In Figure 2.3, the *ReserveRoomHandler* class plays the role of a controller. It coordinates other classes in the realization of the Reserve Room use case. In particular, it has a *makeReservation()* operation to coordinate the actions to make a reservation. The *Room* class plays the role of a resource that can be reserved. It is responsible for retrieving and updating information about the room's availability.

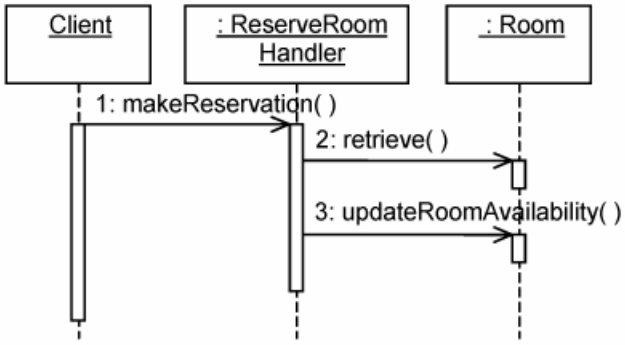

Figure 2.3: Use-Case Realization Example: An Interaction Diagram for the *Reserve Room* Use-Case Realization [2]

From *Reserve Room* use-case realization event flows, the *ReserveRoomHandler* class is specific to this use case, but the *Room* class might be used in other use cases. Therefore, we put the *Room* class into aspect. Figure 2.4 shows the use-case slice of *Reserve Room* use case. This use-case slice contains the following:

1. Collaboration. The collaboration contains a set of diagrams that describe how the *Reserve Room* is realized.
2. Specific Classes. The *ReserveRoomHandler* class is specific to this use-case realization.
3. Specific Extensions. The *Room* class is needed by several use-case realizations. However, the *retrieve()* and *updateAvailability()* are specific to the *Reserve Room* use-case realization. This is defined within a class extension in the use-case slice.

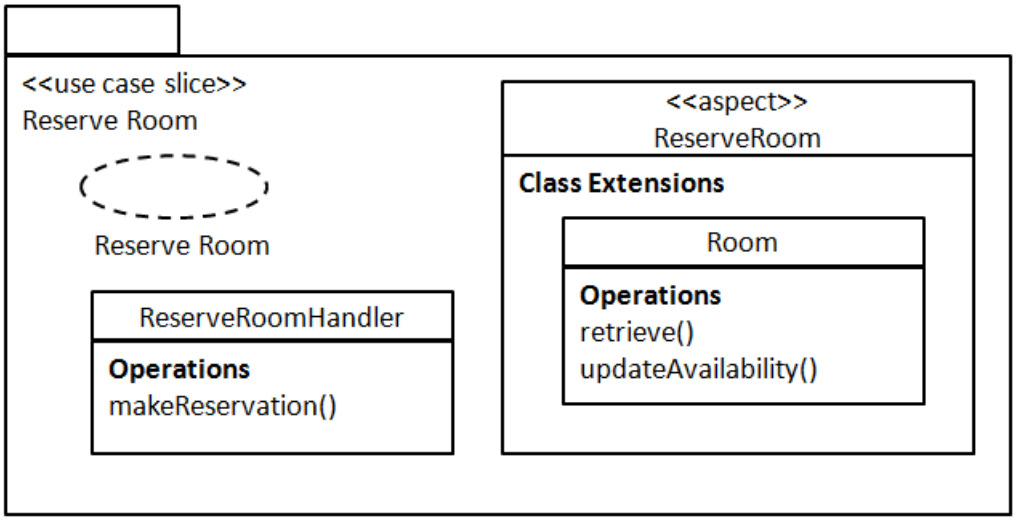

Figure 2.4: Use-Case Slice Example: *Reserve Room* Use-Case Slice [2]

However, some classes are part of the problem domain, and they are used in many use-case realizations. For example, if we realize another use case, Check In Customer use case. The event flows of this use case are shown in Figure 2.5. In figure 2.5 shows that there is the retrieve() operation in the Room class, that is the same as in Reserve Room use-case realization. Therefore, we should put this duplicate part in other containment in order to reuse this part. This containment is called non-use-case-specific Slice.

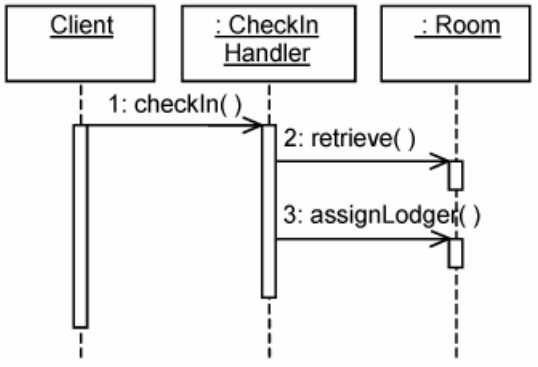

Figure 2.5: Another Use-Case Realization Example: An Interaction Diagram for the *Check In Customer* Use-Case Realization [2]

A non-use-case-specific slice is different from a use-case slice in that it contains no aspects. This is because it defines a base and does not need to add to any existing classes. Non-use-case-specific slice will be extended by other use-case slices. The example of non-use-case-specific slice is shown in Figure 2.6. In Figure 2.6, Room class's retrieve()

operation is used by Reserve Room use case and Check In Customer use case, so it is put in Hotel Management non-use-case-specific slice. Then, the two use-case slices extend Hotel Management slice.

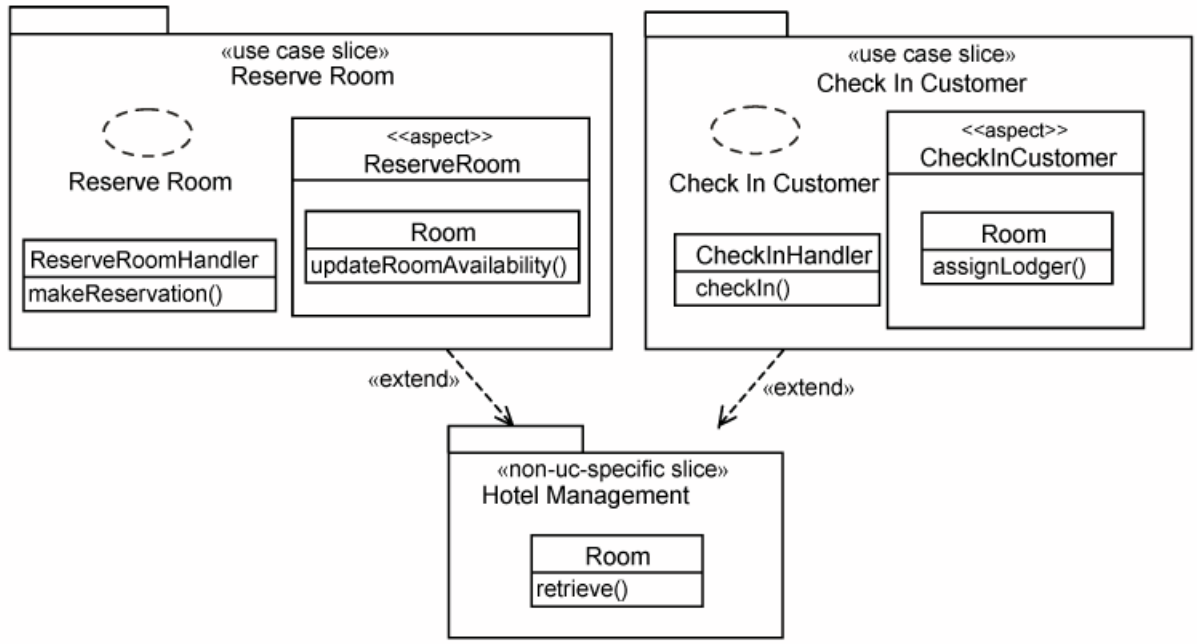

Figure 2.6: Non-Use-Case-Specific Slice Example [2]

2.2.2 Aspect-Oriented Software Development Process and Artifacts

The application of use case concept to AOSD makes the process of AOSD almost similar to UDSD but the artifacts are different in some phases. In order to create the artifacts in systematic way, UML paradigm is applied to AOSD. In AOSD, development process can be divided into five phases in the same way as UDSD. The five phases of AOSD process are as follows;

- Requirements

  This phase is the same as requirements phase in UDSD. Developers capture requirements from customer 's needs using use case model. The major artifacts of requirements phase are use case diagram and use case description. The artifacts are the same as UDSD.

- Analysis

  Based on each use case, developers create use case realization which consists of static and dynamic behavior of the use case. In AOSD, instead of class diagram,

developers use use-case slices to represent static behavior of use case. Consequently, the major artifacts of this phase are use-case slice and collaboration diagram from each use case.

- Design

  In design, developers shape the system and find its form that lives up to all requirements including nonfunctional requirements and platform-specific constraints. The artifacts of this phase are based on analysis ' artifacts. The major artifacts of this phase are the same as analysis; use-case slice and collaboration diagrams.

- Implementation

  In implementation, developers start with the result from design and implement the system in terms of components, that is, source code, binaries, and so on. The major artifacts of this phase are source codes including AOP technique to the code.

- Test

  In the test workflow, developers verify the result from implementation. In AOSD, there is a new technique to separate test elements and elements being tested from each other. It provides use-case test slice as a tool for test cases design. The major artifacts of this phase are test cases and test results.

All the phases and artifacts of AOSD are shown in Table 2.2. Again, in our research study, we focus on the phases before implementation, those are, requirements, analysis, and design phase. The implementation and test are beyond our scope.

Table 2.2: AOSD process and artifacts

Phase	Artifacts
Requirements	use case diagram and use case description
Analysis	use-case slice and collaboration diagram
Design	use-case slice and collaboration diagram
Implementation	source code including AOP technique
Test	test case and test results

2.3 Metrics and Measurement

Software measurement is a task in software development to define, collect and analyze data on software products or software process in order to extract quantified attribute of a characteristic of a software product or the software process. After software measurement, developers use the measurement results as a motivation to improve software being developed in its products or its process. Moreover, in the early stage of software development, the measure from software measurement can be used as a reference in decision making.

In software measurement task, we use not only one but several metrics to achieve measurement goals. Software metric is a function that has input and output. It has software data as inputs and a single numeric value as output. The output is interpreted as the degree to which software possesses a given attributes that affects its quality [5].

In order to successfully apply the metrics and measurement to the software system, we have to choose good metrics. Good metrics should facilitate the development of models that are capable of predicting process or product parameters, not just describing them. Thus, ideal metrics should be: [6]

- Simple, precisely definable so that it is clear how the metric can be evaluated.
- Objective, when different people perform the same measurement, all of the values should be the same.
- Easily obtainable (for example, at reasonable cost)
- Valid, the metric should measure what it is intended to measure.
- Robust, relatively intensive to insignificant changes in the process or product.

In addition, for maximum utility in analytic studies and statistical analyses, metrics should have data values that belong to appropriate measurement scales.

Software metrics can be classified into various categories, according to [6], are as follows;

- Product metrics and process metrics. Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Process metrics, on the other hand, are measures of the software development process, such as overall time, type of methodology used, or the average level of experience of the programming staff.
- Objective metrics and subjective metrics. Objective metrics are measures that always result in identical values for a given metric, as measured by two or more qualified observers. Subjective metrics are measures that even qualified observers may measure different values for a given metric, since their subjective judgment is involved in arriving at the measured value.
- Primitive metrics and computed metrics. Primitive metrics are those that can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some manner from other metrics.

In our research study, the metric suites, that we used to evaluate the system implemented by UDSD and AOSD, are product metrics. Since, the process of UDSD and the process of AOSD are similar. Their processes both consist of requirements, analysis, design, implementation, and test because AOSD process has been developed from UDSD

process. Consequently, we would rather compare the products of these two approaches than their processes. Moreover, our metric suites are objective because our metrics have been defined in mathematical terms, so the observers can apply these metrics as many times with the same results. Lastly, our metrics are computed metrics because we computed from number of components and their relationships.

Chapter 3

Related Works

This chapter presents some of the research work has been done on the metrics that measure separation of concerns.

3.1 Separation of Concerns Metrics

In our research study, we focus on the reduction of crosscutting concerns and increasing of maintainability and reusability of AOSD comparing to UDSD. In software measurement community, there are some metrics defined to measure the separation of concerns of software systems. Separation of concerns refers to the ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern [13].

3.1.1 Sant'Anna C.'s Metrics

Sant 'Anna C. et al. proposed the metrics suite to capture information about design and code in terms of fundamental software attributes of aspect-oriented systems, such as separation of concerns, coupling, cohesion and size. In the coupling, cohesion and size aspects, the metrics were defined from CK metrics because the CK metrics are based on a sound measurement theory and have been widely used and empirically validated. In the separation of concerns aspect, there are three metrics defined as follows [14]:

- Concern Diffusion over Components (CDC). CDC is a design metric that counts the number of primary components whose main purpose is to contribute to the implementation of a concern. Furthermore, it counts the number of components that access the primary components by using them in attribute declarations, formal parameters, return types throws declarations and local variables, or call their methods.

  The higher values of CDC metric means the more components contribute to fulfill a given concern. One concern should not scatter to too many components. Therefore, low values of CDC metric are desirable.

- Concern Diffusion over Operations (CDO). CDO counts the number of primary operations whose main purpose is to contribute to the implementation of a concern. In addition, it counts the number of methods and advices that access any primary component by calling their methods or using them in formal parameters, return types, throws declarations and local variables. Constructors also are counted as operations.

  In the same way as CDC, the higher CDO metric means the more operations contribute to fulfill a given concern. Therefore, the low values of CDO metric are desirable.

- Concern Diffusion over LOC (CDLOC). CDLOC counts the number of transition points for each concern through the lines of code. Transition points are the point in the code where there is a transition from the lines of code that do not implement a given concern to the lines of code that implement a given concern.

  The lower the CDLOC, the more localized is the concern code. Therefore, low values of CDLOC metric are desirable.

Although these metrics can measure separation of concerns of software systems, but this is just one aspect of this problem. CDC and CDO metric can measure how much a given concern diffuse to the systems in terms of components and operations. However, they do not measure concerns that cut across each other. This causes a problem that concerns cannot be separated from each other. Consequently, we have to measure this aspect of the separation of concerns.

Again, in our research study, we focus on the products before implementation phase, but CDLOC metric is measured from source code, so this metric is beyond our scope.

3.1.2 Conejero J.'s Metrics

Conejero J. et al. proposed metrics for crosscutting concerns as predictors of software instability. The problem of crosscutting concerns is usually described in terms of scattering and tangling. Scattering occurs when the realization of a concern is spread over the software modules whilst tangling occurs when the concern realization is mixed with other concerns in a module [15]. In this research, three sets of metrics were defined; metrics for scattering, metrics for tangling, and metrics for crosscutting concerns. These metrics were used to measure the crosscutting of concerns of software systems in requirements phase. The use case descriptions were used as the materials to be measured.

In our research, we focus on the reduction of crosscutting concerns of AOSD comparing to UDSD. These three sets of metrics can be used to identify the scattering, tangling, and crosscutting concerns attributes of software systems. Therefore, we apply these metrics to our research. However, as mentioned in Section 2.1.1 and 2.2.2, the requirements phase of UDSD and the requirements phase AOSD have the same process and create the same products. In this phase, we define the use case model from requirement specification and describe each use case in use case descriptions. Consequently, in order to compare system implemented by UDSD and system implemented by AOSD, we apply these metrics to the

use-case realization; analysis and design phase. The definition and mechanism of these metrics will be explained in more detail in Section 4.2.2.

Chapter 4

Our Approach

This chapter presents the approach that we use in our research in order to achieve our objectives; to evaluate of AOSD complexity in the comparison to UDSD complexity to see how much AOSD can help improve maintainability in UDSD and how much AOSD can help reduce the effect of crosscutting concerns, which can be divided into scattering and tangling. First, we describe about how to compare these two approaches. Then, we explain about metrics suite used in our research; change impact metrics suite and crosscutting concerns metrics suite.

4.1 How to Compare UDSD and AOSD

For the two systems implemented by the same approach, we can apply metrics to measure both of them without any transformations or rules to compare and then can compare the measures of the same metrics easily. On the other hand, in order to compare systems implemented by different approaches, we need mechanisms or rules to normalize them into the same level of abstraction.

Although, UDSD and AOSD with use cases have been developed from the same background theory, but there are some differences between these two approaches. The differences occur because AOSD added new constructs to the systems. Those constructs are aspects and their elements; advices and intertype declarations. In our research study, we have to find the way to compare systems implemented by UDSD and systems implemented by AOSD in a consistent and meaningful manner.

Figueiredo, E. et al. proposed a generic concern-oriented meta-model of the structural abstractions defined for aspect-oriented system [16] as shown in Figure 4.1. It not only defines possible relations of concerns and the system's structure, but also subsumes key abstractions for module specifications. Each type of abstraction is alternatively called an element. Concerns can be realized by an arbitrary set of elements. For the clarification, in this meta-model, they used the arrow with diamond rectangle and the arrow without diamond rectangle as a dependency between elements. This diamond rectangle does not mean the aggregation defined in UML model, but the arrow with diamond rectangle means one-to-many relationship and the arrow without diamond rectangle means one-to-

one dependency. An aspect-oriented system S consists of a set of components, denoted by *C(S)*. A components c has an interface, *I(c)*. Besides, each component c consists of a set of attributes, *Att(c)*, a set of operations, *Op(c)*, and a set of declaration, *Dec(c)*. The set of members of a component c is defined by $M(c) = Att(c) \cup Op(c) \cup Dec(c)$.

For generality purposes, a component is a unified abstraction to both aspectual and non-aspectual modules. This decision makes the meta-model paradigm and language independent.

An operation o consists of a return type, *Rt(o)*, a set of parameters, *Par(o)*, a pointcut expression, *PE(o)*, and a set of statements, *St(o)*. A declaration d can also have a pointcut expression, *PE(d)*.

On top of the structure, we can define concerns. A concern is not an abstraction of a modeling or programming language, such as components and operations. However, a concern can be considered as an abstraction which is addressed by those elements that have the purpose of realizing it.

The set of concerns addressed by the system S is defined as *Con(S)*. Furthermore, a concern con can be realized by a set of components, *C(con)*, a set of attributes, *Att(con)*, a set of operations, *Op(con)*, or a set of declarations, *Dec(con)*. The set of members that implement a concern con is defined as $M(con) = Op(con) \cup Att(con) \cup Dec(con)$.

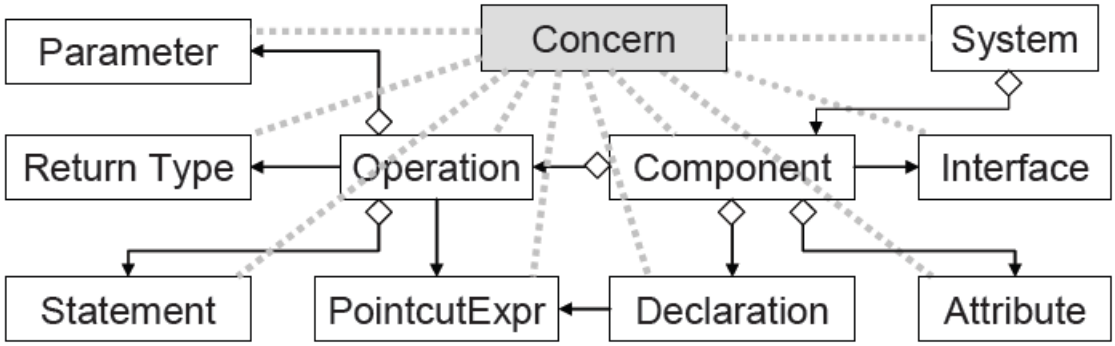

Figure 4.1: Abstract Meta-Model of Aspect-Oriented Systems [16]

This structure is abstract enough to be instantiated for different modeling and programming languages. In our research, we focus on the systems implemented by UDSD and systems implemented by AOSD. Therefore, in order to compare these two approaches in a consistent and meaningful manner, we instantiate this meta-model structure for UDSD and AOSD as shown in Table 4.1. In Table 4.1, we describe the instantiations of each element in the meta-model for UDSD and AOSD approach. The instantiation for system element is UDSD system and AOSD system for UDSD and AOSD respectively. For concern element, we refer to use case in both approaches because the use-case technique provides the means to systematically model stakeholder concerns by walking through meaningful interactions between end users and the system [2]. Therefore, use case can refer concern from stakeholder. The component in UDSD is class or interface. But in AOSD, there are class, interface and aspect as components. The interface in both UDSD

and AOSD is method signature. The attributes in UDSD are class variable and field. In AOSD, intertype attribute has been added to be an attribute of the system. The operation in UDSD is either method or constructor. In AOSD, there are method, constructor, intertype method, intertype constructor, and advice as operations. The declaration appears only in AOSD systems. The declaration in AOSD is either pointcut or declare statement.

Table 4.1: Meta-Model Instantiation for UDSD and AOSD Systems

Element	UDSD	AOSD
System	UDSD System	AOSD System
Concern	Use Case	Use Case
Component	Class and Interface	Class, Interface, and Aspect
Interface	Method Signature	Method Signature
Attribute	Class Variable and Field	Class Variable, Field, and Intertype Attribute
Operation	Method and Constructor	Method, Constructor, Intertype Method and Constructor, and Advice
Declaration	-	Pointcut and Declare Statement

4.2 Our Metrics Suites for Evaluating UDSD and AOSD

After we define how to compare the UDSD and AOSD systems, we have to define metrics that can extract the values of attributes of systems in order to see the efficiency of AOSD over UDSD.

4.2.1 Change Impact Metrics Suite

In early stage of our research, we have focused on how AOSD approach improves maintainability of UDSD systems. Maintainability means the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [17]. From this definition, the maintainability directly relates to the change to the system. Therefore, we propose the metrics suite to measure how system is affected by the change.

Change Definition and Type of Change

Software change is inevitable. This is because new requirements emerge, the business environment changes, errors must be repaired, new equipment must be accommodated, or

the performance or reliability may have to be improved. Change to the software systems can be divided into four types, according to [18];

- Addition. New elements are inserted into the base system.
- Removal. An element in the base system is removed.
- Modification. An element has some properties modified.
- Derivation. Elements are refined and/or move to accommodate the changes.

Components and Relationships

In UDSD and AOSD process, there are products created during each phase. In both UDSD and AOSD requirements phase, the use case diagram is created. In UDSD analysis phase, the class diagram and collaboration diagram are created. In AOSD analysis phase, the use case slice and collaboration diagram are created. In UDSD design phase, the products are the same as in its analysis phase but they describe system in more detail about implementation issues. In AOSD design phase, the products are also the same as its analysis phase.

According to the products created in development process, these products are created in form of UML diagrams. In fact, there is the same characteristic amongst those diagrams. The diagram consists of components and relationships between components. In use case diagram, there are use cases as components and association between use cases as relationships. In class diagram, there are classes as components and classes' association as relationships. In use-case slice, there are classes and aspects as components and relationships between class and class, class and aspect and aspect and aspect as relationships. And in collaboration diagram, there are classes and aspects as components and their method calls and operation calls as relationships. We can conclude the components and relationships in each diagram as shown in Table 4.2.

Table 4.2: Component and Relationship of System's Diagrams

Approach	Diagram	Component	Relationship
UDSD	Use Case Diagram	Use Cases	Use Case Associations
	Class Diagram	Classes	Class Associations
	Collaboration Diagram	Classes	Method Calls
AOSD	Use Case Diagram	Use Cases	Use Case Associations
	Use-Case Slice	Classes and Aspects	Associations of Class and Class, Class and Aspect, and Aspect and Aspect
	Collaboration Diagram	Classes and Aspects	Method Calls and Intertype Operation and Advice Calls

Change Impact Metric Definition

When the change occurs, software systems have to be modified. The system S after modifying can be divided into four parts;

- Added Part.

  This part of the system is the part that new components and new relationships are introduced to the system. The components of this part are defined as *Add(c)* and the relationships of this part are defined as *Add(r)*.

- Modified and Derived Part.

  This part of the system is the part that existing components and relationships have to be modified or reorganized because of the change. The components of this part are defined as *Mod(c)* and the relationships of this part are defined as *Mod(r)*.

- Removed Part.

  This part of the system is the part that components and relationships have been removed from the system after modifying the system to deal with change. The components of this part are defined as *Rem(c)* and the relationships of this part are defined as *Rem(r)*.

- No Change Part. This part of the system is the part that is not related to the effect of change. It is not modified or removed from the system according to the change. The components of this part are defined as *Noc(c)* and the relationships of this part are defined as *Noc(r)*.

The change impact metric is a metric that measures how much the system is affected by the change comparing to the whole system.

The impact of change on components *Imp(c)* is defined as;

$$Imp(c) = Add(c) + Mod(c) + Rem(c) \tag{4.1}$$

And the impact of change on relationships *Imp(r)* is defined as;

$$Imp(r) = Add(r) + Mod(r) + Rem(r) \tag{4.2}$$

The components in the entire system *Sys(c)* is defined as;

$$Sys(c) = Add(c) + Mod(c) + Rem(c) + Noc(c) \tag{4.3}$$

The ralationships in the entire system *Sys(r)* is defined as;

$$Sys(r) = Add(r) + Mod(r) + Rem(r) + Noc(r) \tag{4.4}$$

The degree of change impact I is defined as;

$$I = \frac{Imp(c) + Imp(r)}{Sys(c) + Sys(r)} \tag{4.5}$$

As can be seen in the formulas, the removed part is still counted as one part of the system in order to measure the impact of change to the whole system. Because the removed part is counted as one of the change impact, so the entire system has to include this part to the measure. For simplicity, we illustrate the system components and relationships after the change in Figure 4.2.

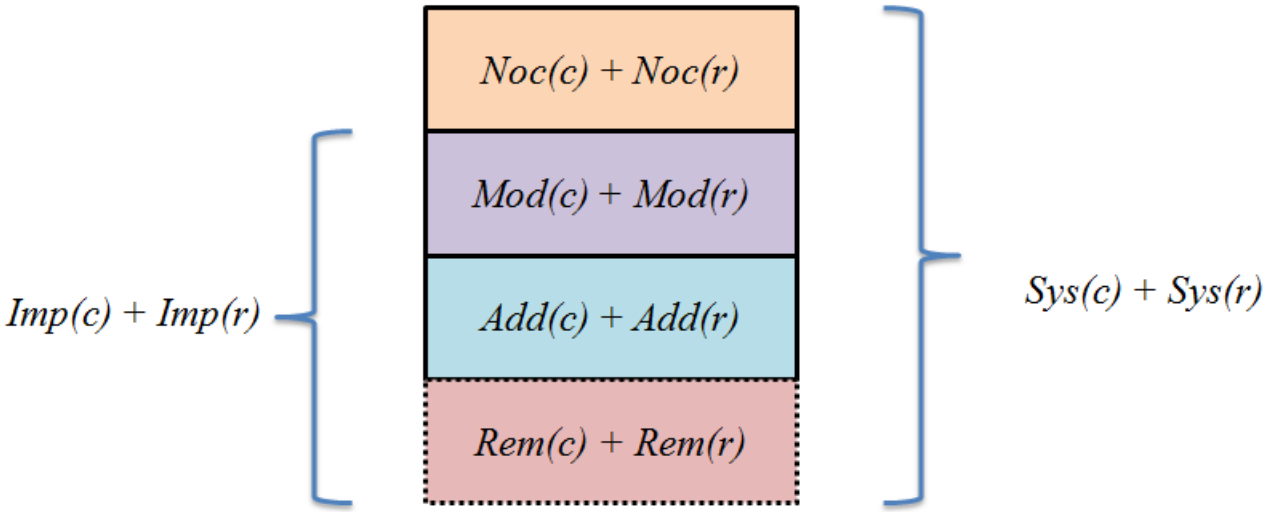

Figure 4.2: System after Change

For the change impact metric, we can apply this metric to measure the impact of change in the level of each diagram defined earlier in Table 4.2 or for the entire system by counting all the components and relationships from every diagram created from requirements, analysis and design phase.

4.2.2 Scattering, Tangling, and Crosscutting Metrics Suite

In our research, one of our goals is to evaluate the reduction of crosscutting concerns in the systems implemented by AOSD approach comparing to the system implemented by UDSD approach. Therefore, we need to apply metrics that directly measure the crosscutting concerns to our research.

A concern is some part of the problem that we want to treat as a single conceptual unit [4]. Sometimes, a concern affects more than one component and a component contains parts of multiple concerns. These situations are called scattering and tangling respectively. These kind of concerns are called crosscutting concern. The consequences caused by these concerns are reduced comprehensibility, ease of evolution and reusability of software artifacts. Accordingly, these concerns should be separated.

Conejero J. et al. proposed metrics suite for crosscutting concerns as predictors of software instability. First, they proposed a conceptual framework for crosscutting as a basis for measurement of crosscutting concerns. Then, they proposed a set of metrics for measuring scattering, tangling and crosscutting concerns [15].

A Conceptual Framework for Crosscutting

A conceptual framework for crosscutting is based on the study of matrices that represent particular features of a traceability relationship between two different domains. These domains are generically called Source and Target. They used the term Crosscutting Pattern to denote the situation of crosscutting as shown in Figure 4.3.

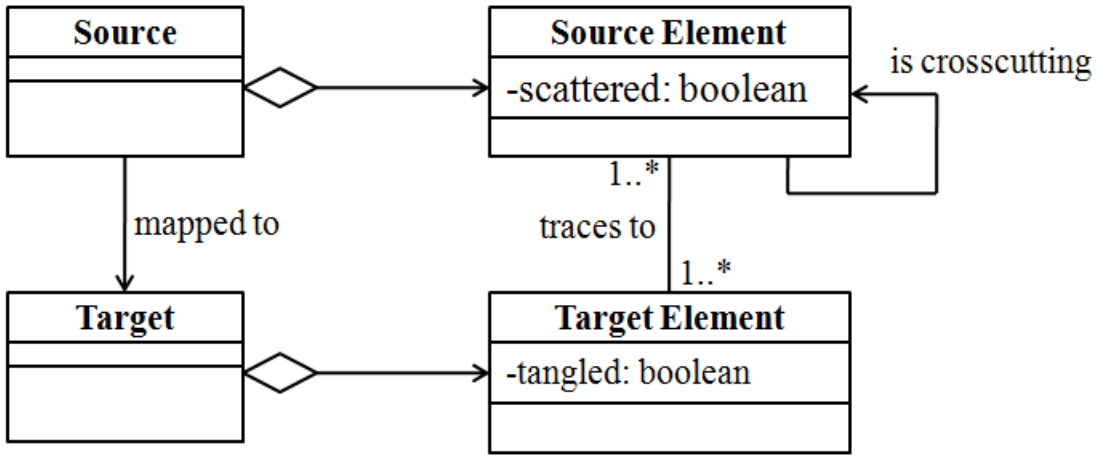

Figure 4.3: Abstract Meta-Model of the Crosscutting Pattern [15]

The relationship between Source and Target can be formalized by two functions f and g, where g can be considered as a special inverse function of f. The two functions were defined as:

$\forall$ s $\in$ Source, f(s)={t $\in$ Target:there exists a trace relation between s and t}

$\forall$ t $\in$ Target, g(t)={s $\in$ Source:there exists a trace relation between s and t}

The concepts of scattering, tangling, and crosscutting are defined as specific cases of these functions.

Defintion 1 [Scattering]: We say that an element s $\in$ Source is scattered if card(f(s))>1, where card refers to cardinality of f(s). In other words, scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements.

Defintion 2 [Tangling]: We say that an element t $\in$ Target is tangled if card(g(t))>1. Tangling occurs when, in a mapping between source and target, a target element is related to multiple source elements.

There is a specific combination of scattering and tangling which it is called crosscutting.

Defintion 3 [Crosscutting]: Let s1 and s2 $\in$ Soruce, s1 $\neq$ s2, we say that s1 crosscuts s2 if card(f(s1))>1 and $\exists$ t $\in$ f(s1): s2 $\in$ g(t). Crosscutting occurs when, in a mapping between source and target, a source element is scattered over target elements and where at least one of these target elements, some other source element is tangled [19].

In this meta-model of the crosscutting, we can instantiate the abstraction of it by defining the specification of the two domains, source and target that has traceability relationship to each other. In our research, we instantiated the abstract meta-model of crosscutting by defining use cases as sources and classes, interfaces and aspects, which for simplicity we call them "modules", as targets. The instantiation is shown in Table 4.3.

Table 4.3: Crosscutting Meta-Model Instantiation for UDSD and AOSD Systems

Element	UDSD	AOSD
Source	Use Case	Use Case
Target	Module (Class and Interface)	Module (Class, Interface, and Aspect)

Identification of Crosscutting

Conejero J. et al. defined the dependency matrix to represent function f. For example, in a software system, there are 5 use cases and 6 modules. In this case, modules mean classes, interfaces and aspects. The dependency of this system is shown in Table 4.4 in order to trace the dependency between use case and class. A 1 in a cell means that the class element of the corresponding column contributes to addresses the use case element of the corresponding row. On the other hand, a 0 means there is no dependency between the class element of the corresponding column and addresses the use case element of the corresponding row.

Table 4.4: Example of Dependency Matrix

		Module					
		m[1]	m[2]	m[3]	m[4]	m[5]	m[6]
Use Case	uc[1]	1	0	0	1	0	0
	uc[2]	1	0	1	0	1	1
	uc[3]	1	0	0	0	0	0
	uc[4]	0	1	1	0	0	0
	uc[5]	0	0	0	1	1	0

Based on this matrix, two different matrices called scattering matrix and tangling matrix are derived. According to the definition of scattering, we focus on how a use case is related to classes. Therefore, the scattering matrix is the same matrix as dependency matrix which it defines use case as rows and module as columns. According to the definition of tangling, we focus on how a module is related to use cases. Therefore, the tangling

matrix is the transpose of dependency matrix which it defines module as rows and use case as columns. The example of tangling matrix for dependency matrix in Table 4.4 is shown in Table 4.5.

Table 4.5: Example of Tangling Matrix

		Use Case				
		uc[1]	uc[2]	uc[3]	uc[4]	uc[5]
Module	m[1]	1	1	1	0	0
	m[2]	0	0	0	1	0
	m[3]	0	1	0	1	0
	m[4]	1	0	0	0	1
	m[5]	0	1	0	0	1
	m[6]	0	1	0	0	0

The crosscutting product matrix is obtained through the multiplication of scattering matrix and tangling matrix. The crosscutting product matrix shows the quantity of crosscutting relations as shown in Table 4.5. In Table 4.5, we show the result of multiplication of scattering matrix in Table 4.3 and tangling matrix in Table 4.4. This matrix is used to derive the final crosscutting matrix as shown in Table 4.6. A cell in the final crosscutting matrix denotes the occurrence of crosscutting, but abstracts the quantity of crosscutting. In the crosscutting matrix, the diagonal cells are set to be zero because a use case cannot crosscut itself.

Table 4.6: Example of Crosscutting Product Matrix

		Use Case				
		uc[1]	uc[2]	uc[3]	uc[4]	uc[5]
Use Case	uc[1]	2	1	1	0	1
	uc[2]	1	3	1	1	1
	uc[3]	0	0	0	0	0
	uc[4]	0	1	0	1	0
	uc[5]	1	1	0	0	2

In our research, in order to create dependency matrix, we have to consider the artifacts that can trace the relationship between use case and module. Therefore, we use the union of all class diagrams from all use cases as material for dependency matrix in UDSD and the union of all use-case slices from all use cases as material in AOSD.

Metrics for Scattering

According to the definition of scattering, NScattering of a use case element sk is the number 1 ’s in the corresponding row (k) of the dependency matrix:

Table 4.7: Example of Crosscutting Matrix

		Use Case				
		uc[1]	uc[2]	uc[3]	uc[4]	uc[5]
Use Case	uc[1]	0	1	1	0	1
	uc[2]	1	0	1	1	1
	uc[3]	0	0	0	0	0
	uc[4]	0	1	0	1	0
	uc[5]	1	1	0	0	0

$$NScattering(s_k) = \sum_{j=1}^{|T|} dm_{kj} \tag{4.6}$$

Where $|T|$ is the number of module elements and dm_{kj} is the value of the cell [k,j] of the scattering matrix. This metric measures how scattered a use case is. This *NScattering* metric can be normalized in order to obtain a value between 0 and 1. Then, *Degree of scattering* of the use case element s_k is defined as:

$$Degree\ of\ scattering(s_k) = \begin{cases} \frac{\sum_{j=1}^{|T|} dm_{kj}}{|T|} & \text{if } \sum_{j=1}^{|T|} dm_{kj} > 1 \\ 0 & \text{if } \sum_{j=1}^{|T|} dm_{kj} = 1 \end{cases} \tag{4.7}$$

The closer to zero this metric is, the better encapsulated the use case element. On the other hand, when the metric has a value closer to 1, the use case element is highly spread over the module elements and it is worse encapsulated. In order to have a global metric for how much scattering the system 's use cases are, the concept of *Global scattering (GScattering)* were defined which is obtained by calculating the average of the *Degree of scattering* values for each use case element:

$$GScattering = \frac{\sum_{i=1}^{|S|} Degree\ of\ scattering(s_i)}{|S|} \tag{4.8}$$

Where $|S|$ is the number of analyzed use case elements.

Metrics for Tangling

Similarly to *NScattering* for scattering, *NTangling* metric for the module element t_k are defined, where $|S|$ is the number of use case elements and dm_{ki} is the value of the cell [k,i] of the dependency matrix:

$$NTangling(t_k) = \sum_{i=1}^{|S|} dm_{ik} \tag{4.9}$$

This metric measures the number of use case element addressed by a particular module element.

Similarly to the steps performed for the scattering metrics and two tangling metrics were defined: *Degree of tangling* and *GTangling*. These metrics represent the normalized tangling for the module element t_k and the global tangling, respectively:

$$Degree\ of\ tangling(t_k) = \begin{cases} \frac{\sum_{i=1}^{|S|} dm_{ik}}{|T|} & \text{if } \sum_{i=1}^{|S|} dm_{ik} > 1 \\ 0 & \text{if } \sum_{i=1}^{|S|} dm_{ik} = 1 \end{cases} \tag{4.10}$$

$$GTangling = \frac{\sum_{j=1}^{|T|} Degree\ of\ tangling(t_j)}{|T|} \tag{4.11}$$

The *Degree of tangling* metric may take values between 0 and 1, where he value 0 represents a module element addressing only one use case element. The number of use case elements addressed by the module element increases as the metric is closer to 1.

Metrics for Crosscutting

Metrics for crosscutting can be divided into three metrics: *Crosscutpoints*, *NCrosscut* and *Degree of crosscutting*. These metrics are extracted from the crosscutting product matrix and the crosscutting matrix.

The *Crosscutpoints* metric is defined for a use case element s_k as the number of module elements where s_k is crosscutting to other source elements. This metric is calculated from the crosscutting product matrix. The *Crosscutpoints* metric for s_k corresponds to the value of the cell in the diagonal of the row k or, in other words, the cell [k,k] ($ccpm_{kk}$) in the crosscutting product matrix.

$$Crosscutpoints(s_k) = ccpm_{kk} \tag{4.12}$$

The *NCrosscut* metric is defined for the use case element s_k as the number of use case elements crosscut by s_k. The *NCrosscut* metric for s_k is calculated by the addition of all cells of the row k in the crosscutting matrix:

$$NCrosscut(s_k) = \sum_{i=1}^{|S|} ccm_{ki} \tag{4.13}$$

From the *Crosscutpoints* metric and *NCrosscut* metric, the *Degree of crosscutting* metric of a use case element s_k is defined. *Degree of crosscutting* is normalized between 0 and 1, so that those use case elements with lower values for this metric are the best modularized.

$$Degree\ of\ crosscutting(s_k) = \frac{Crosscutpoints(s_k) + Concerns\ crosscut(s_k)}{|S| + |T|} \tag{4.14}$$

To sum up, all the metrics for scattering, tangling, and crosscutting are summarized in Table 4.7.

Table 4.8: Summary of the Scattering, Tangling, and Crosscutting Metrics

Metric	Definition	Relation with matrices	Calculation
NScattering (s_k)	Number of module elements addressing use case element s_k	Addition of the values of cells in row k in dependency matrix (dm)	$= \sum_{j=1}^{\|T\|} dm_{kj}$
Degree of scattering (s_k)	Normalization of $NScattering(s_k)$ between 0 and 1		$= \begin{cases} \frac{\sum_{j=1}^{\|T\|} dm_{kj}}{\|T\|} & \text{if } \sum_{j=1}^{\|T\|} dm_{kj} > 1 \\ 0 & \text{if } \sum_{j=1}^{\|T\|} dm_{kj} = 1 \end{cases}$
GScattering (s_k)	Average of *Degree of scattering* of the use case elements		$= \frac{\sum_{i=1}^{\|S\|} Degree\ of\ scattering(s_i)}{\|S\|}$
NTangling (t_k)	Number of use case elements addressed by module element t_k	Addition of the values of cells in column k in dependency matrix (dm)	$= \sum_{i=1}^{\|S\|} dm_{ik}$
Degree of tangling (s_k)	Normalization of $NTangling(t_k)$ between 0 and 1		$= \begin{cases} \frac{\sum_{i=1}^{\|S\|} dm_{ik}}{\|T\|} & \text{if } \sum_{i=1}^{\|S\|} dm_{ik} > 1 \\ 0 & \text{if } \sum_{i=1}^{\|S\|} dm_{ik} = 1 \end{cases}$
GTangling (t_k)	Average of $Degreeoftangling$ of the module elements		$= \frac{\sum_{j=1}^{\|T\|} Degree\ of\ tangling(t_j)}{\|T\|}$
Crosscut points (s_k)	Number of module elements where the use case element s_k crosscuts to other use case elements	Diagonal cell of row k in the crosscutting product matrix (ccpm)	$= ccpm_{kk}$
NCrosscut (s_k)	Number of use case elements crosscut by the use case element s_k	Addition of the values of cells in row k in the crosscutting matrix (ccm)	$= \sum_{i=1}^{\|S\|} ccm_{ki}$
Degree of crosscutting (s_k)	Addition of the two last metrics normalized between 0 and 1		$= \frac{ccpm_{kk} + \sum_{i=1}^{\|S\|} ccm_{ki}}{\|S\| + \|T\|}$

Chapter 5

Empirical Study and Results

This chapter presents the case study and the results to validate our metrics. First, we describe about the ATM system that is chosen to be our case study and how it has been developed to be ready to be measured. Then, we show our results of the measurement of each metric defined earlier, the comparison between UDSD and AOSD and statistical analysis of our results. Last, we discuss about the characteristic of AOSD that has an effect on our results.

5.1 Case Study: ATM System

In order to validate our metrics that are defined earlier, we have to apply these metrics to some system. In our research, we apply our metrics to ATM system which is introduced in "Designing Concurrent, Distributed, and Real-Time Applications with UML" by Hassan Gomaa [20]. We have chosen this system because it has well-defined requirements specification. Therefore, we can build our system implemented by UDSD and by AOSD in a consistent way.

5.1.1 Problem Description

The problem description for ATM system is defined as follow:

A bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card reader, a cash dispenser, a keyboard/ display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either a checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct

PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.

If the PIN is validated satisfactorily, the customer is prompted for a withdrawal, query, or transfer transaction. Before a withdrawal transaction can be approved, the system determines that sufficient funds exist in the requested account, that the maximum daily limit will not be exceeded, and that there are sufficient funds at the local cash dispenser. If the transaction is approved, the requested amount of cash is dispensed, a receipt is printed containing the information about the transaction, and the card is ejected. Before a transfer transaction can be approved, the system determines that the customer has at least two accounts and that there are sufficient funds in the account to be debited. For approved query and transfer requests, a receipt is printed and the card is ejected. A customer may cancel a transaction at anytime; the transaction is terminated and the card is ejected. Customer records, account records, and debit card records are all maintained at the server.

An ATM operator may start up and close down the ATM to replenish the ATM cash dispenser and for routine maintenance. It is assumed that functionality to open and close accounts and to create, update, and delete customer and debit card records is approved by an existing system and is not part of this problem.

5.1.2 Use Case Model

In the requirements phase, we create the use case model in the same way for both UDSD and AOSD. First, we define use cases from problem description. Then, we describe each use case in more detail in use case description.

According to the problem description of ATM system, five use cases are defined;

- *Validate PIN* use case. This use case describes an event that the system validates customer PIN.
- *Withdraw Funds* use case. This use case describes an event that a customer withdraws specific amount of funds from a bank account.
- *Query Account* use case. This use case describes an event that a customer receives the balance of a bank account.
- *Transfer Funds* use case. This use case describes an event that a customer transfers funds from a bank account to another.
- *Cancel Transaction* use case. This use case describes an event that a customer dismisses current transaction.

All the use cases are defined in use case diagram as shown in Figure 5.1. The detail of each use case is described in use case descriptions in Appendix A.

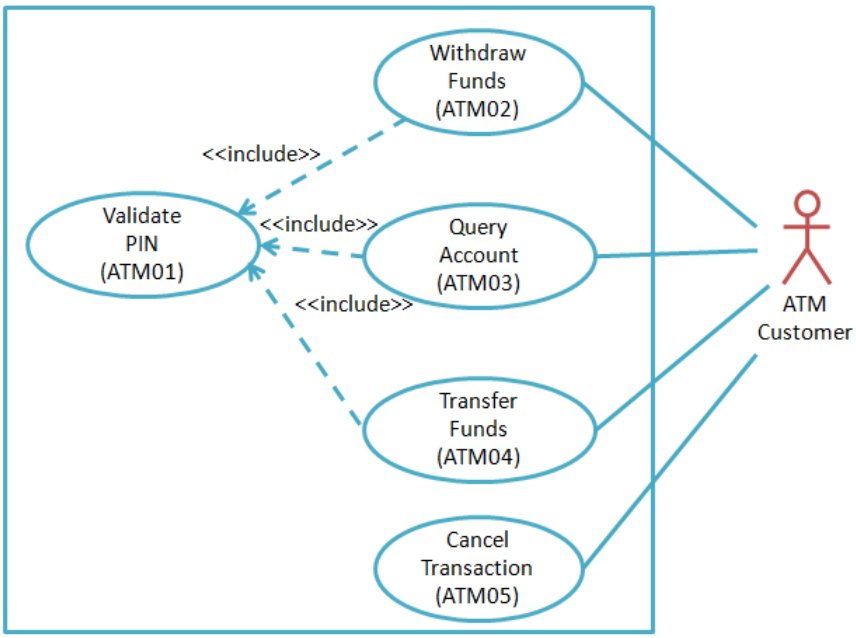

Figure 5.1: ATM System Use Case Diagram

5.1.3 Analysis Model and Design Model

In analysis phase, the process is different between UDSD and AOSD. Therefore, we have to develop the system separately. In UDSD, we create class diagram and collaboration diagram. But in AOSD, we create use-case slice and collaboration diagram based on use-case slice.

UDSD Analysis Model

In UDSD analysis phase, we create class diagrams and collaboration diagrams as use case realizations for each use case. The *Validate PIN* use case is used to exemplify how the use case has been analyzed. In order to realize a use case, we have to consider the flow of events in that use case by using the use case description. In Figure 5.2, class diagram of *Validate PIN* use case is shown. It shows all the classes, their methods, and their attributes necessary for realizing the *Validate PIN* use case.

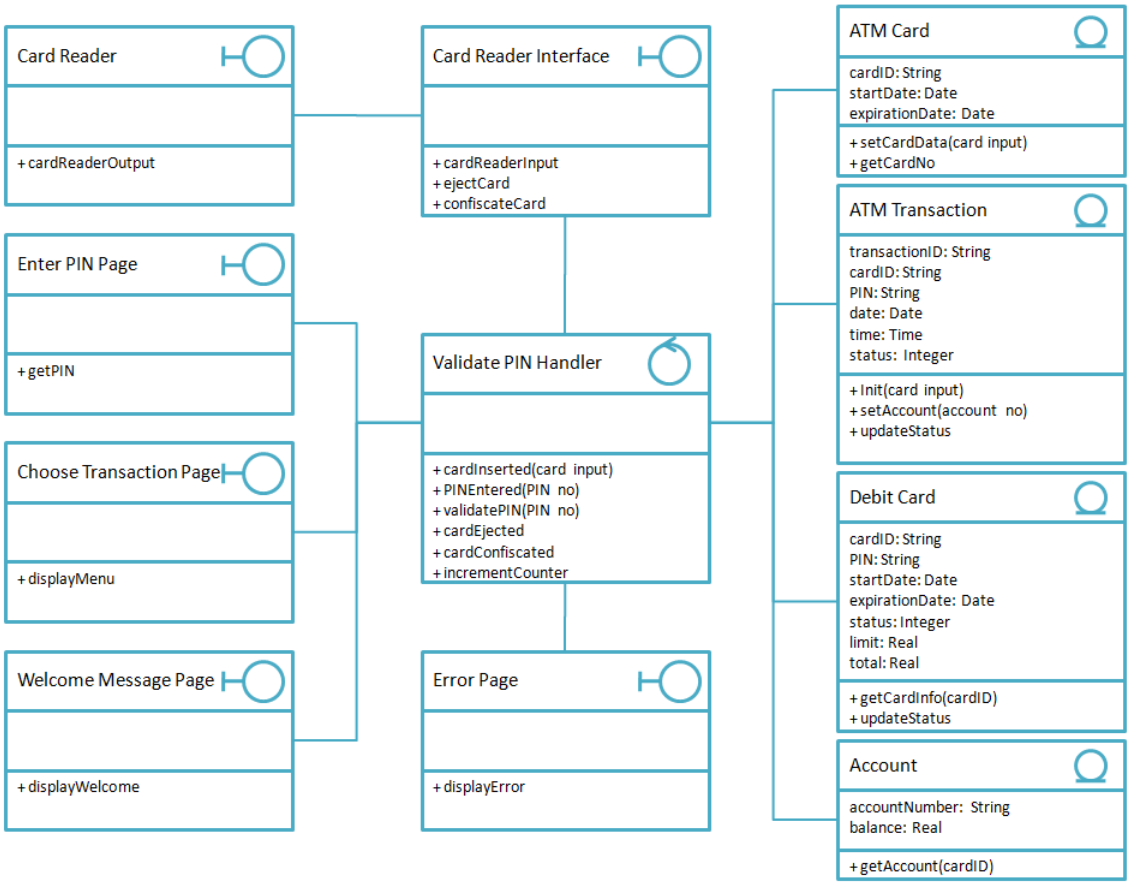

Figure 5.2: Example of UDSD Analysis Model - Class Diagram for *Validate PIN* Use Case

According to the events flow in use case description, we can derive the collaboration of the Validate PIN use case. In Figure 5.3, the collaboration of typical flows of the Validate PIN use case is shown. Note that, we have to consider not only the typical flows but also the alternate flows.

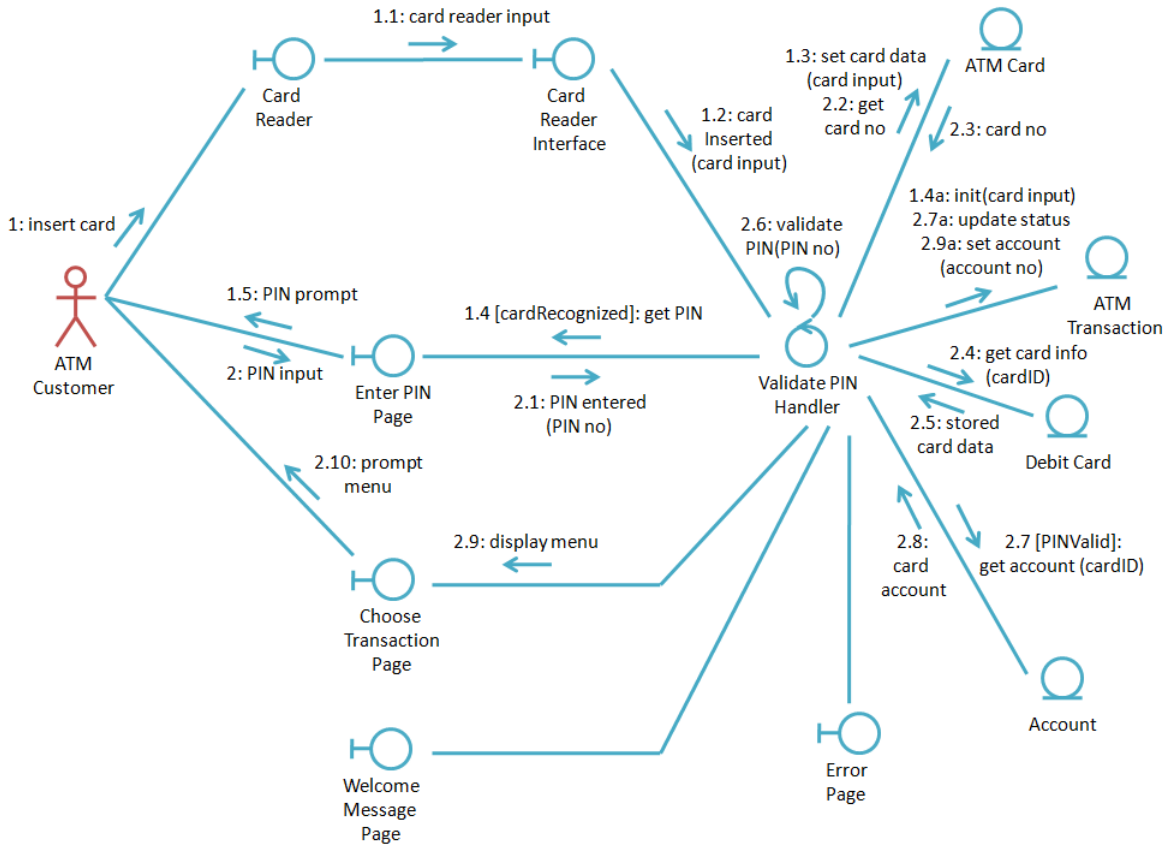

Figure 5.3: Example of UDSD Analysis Model - Collaboration Diagram for *Validate PIN* Use Case Typical Flow

AOSD Analysis Model

In AOSD analysis phase, we create use-case slice and collaboration diagrams as use case realizations for each use case. In order to realize a use case, we have to consider the flow of events in that use case by using the use case description. In Figure 5.4, use case slice of *Validate PIN* use case is shown. It shows all the classes, the aspects, methods, intertype methods and attributes necessary for realizing the *Validate PIN* use case.

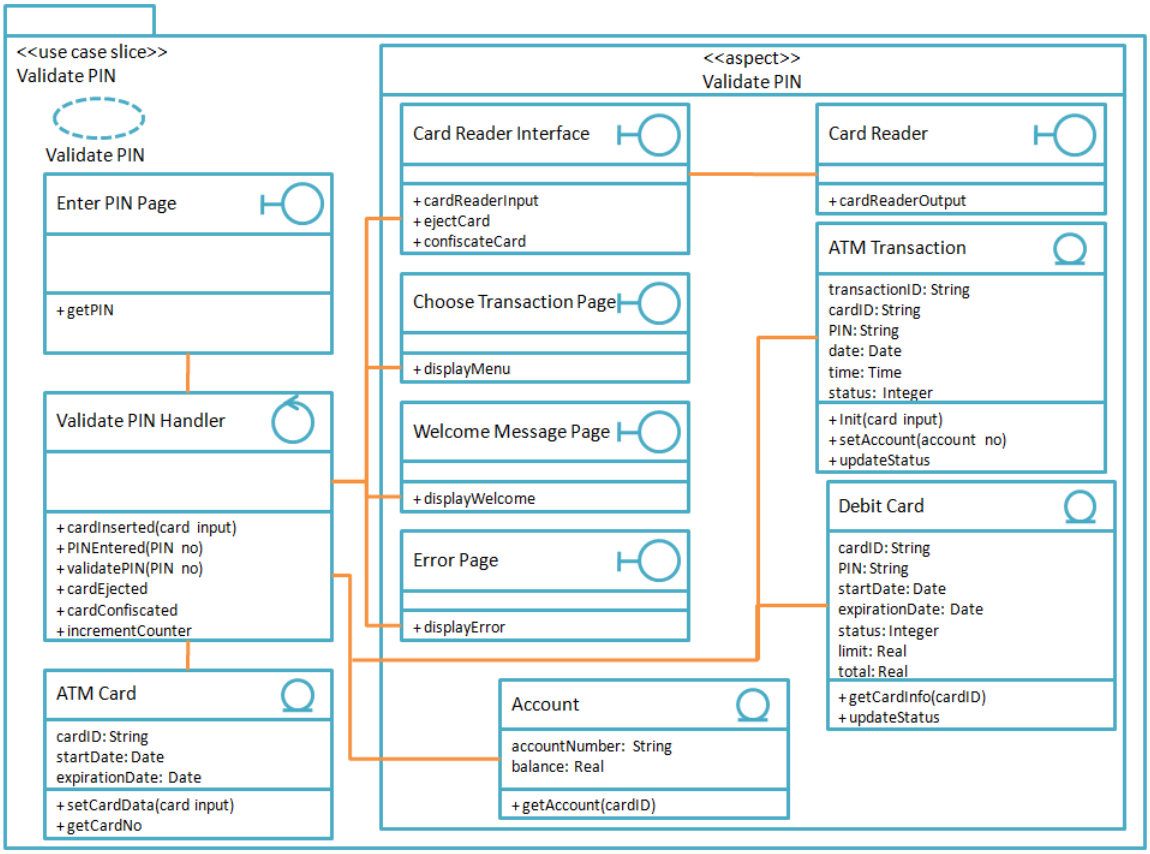

Figure 5.4: Example of AOSD Analysis Model - Use-Case Slice for *Validate PIN* Use Case

However, after realizing all the use cases in the system, there are some parts of classes, especially methods, which can be used by many use cases. Therefore, we put these parts of classes in the non-use-case-specific slice as shown in Figure 5.5. In this Figure, we can see that input and output devices of ATM system can be used amongst many use cases, so we put them in *I/O Handler* slice. Similarly to in *I/O Handler* slice, *Customer Interface* slice (slice interacts with customer), *Bank Data Management* slice (slice handles data in banking system), and *ATM Data Management* slice (slice handles data in ATM) are used in many use cases, so we make them as separate slice.

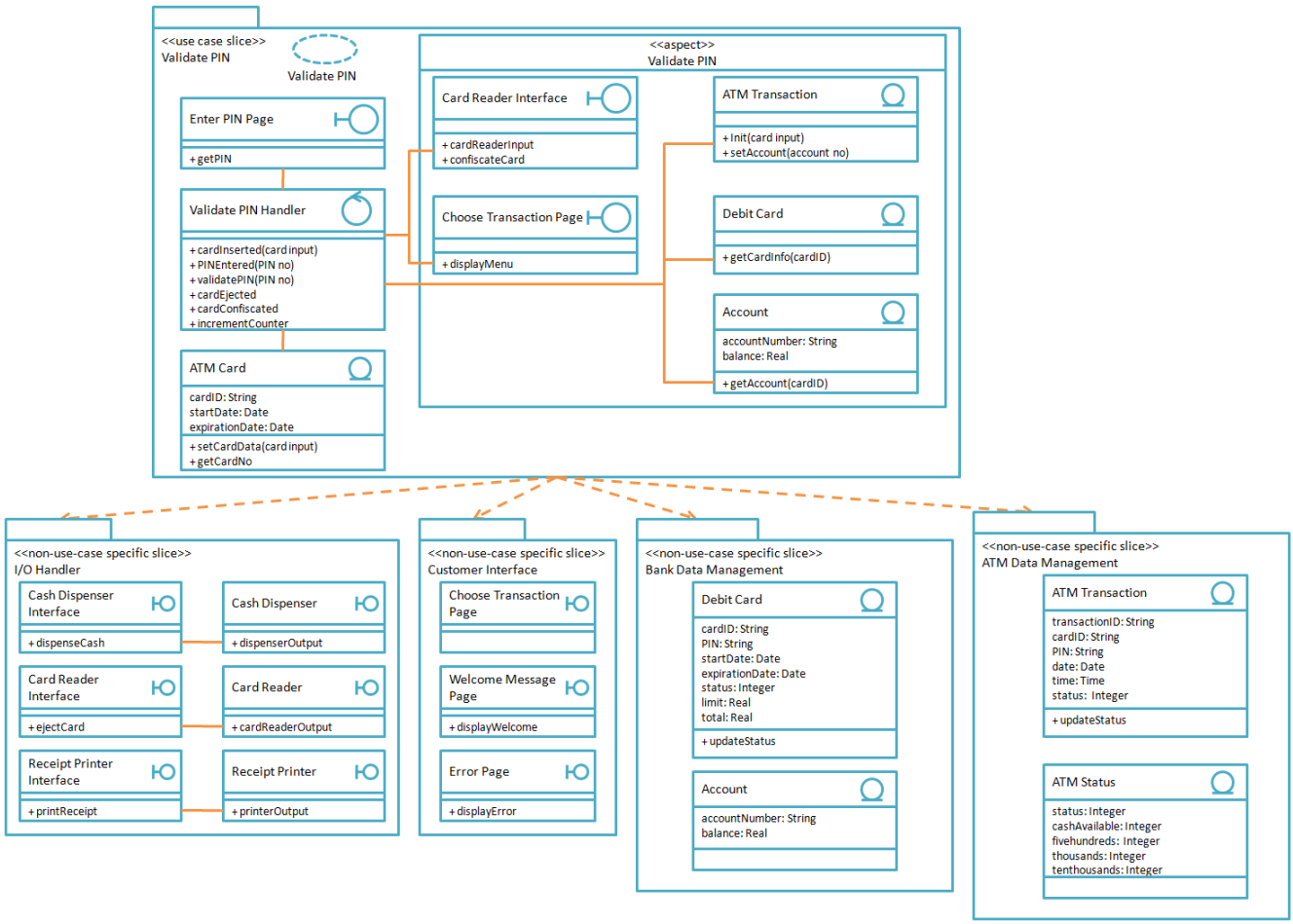

Figure 5.5: Example of AOSD Analysis Model - Use-Case Slice with Non-Use-Case-Specific Slice for *Validate PIN* Use Case

For the collaboration diagram, the method calls are not changed from UDSD to AOSD. But the containments of methods are changed to be aspects for some methods of classes. For the classes which are contained in non-use-case-specific slice, the class structures remain the same. The collaboration diagram for *Validate PIN* use case typical flows in analysis phase of AOSD is shown in Figure 5.6. Note that, we have to consider not only the typical flows but also the alternate flows. In this Figure, there are some methods that belong to *Validate PIN* aspect. Therefore, when some classes need to call these methods, they have to call to *Validate PIN* aspect. For the *ATM Transaction* class, there are methods; *init(cardInput)* and *setAccount(accountNo)* that belong to *Validate PIN* aspect and a method; *updateStatus()* that belongs to original *ATM Transaction* class but it is put in *ATM Data Management* slice. Therefore, the *updateStatus()* method is still directly called from the *ATM Transaction* class.

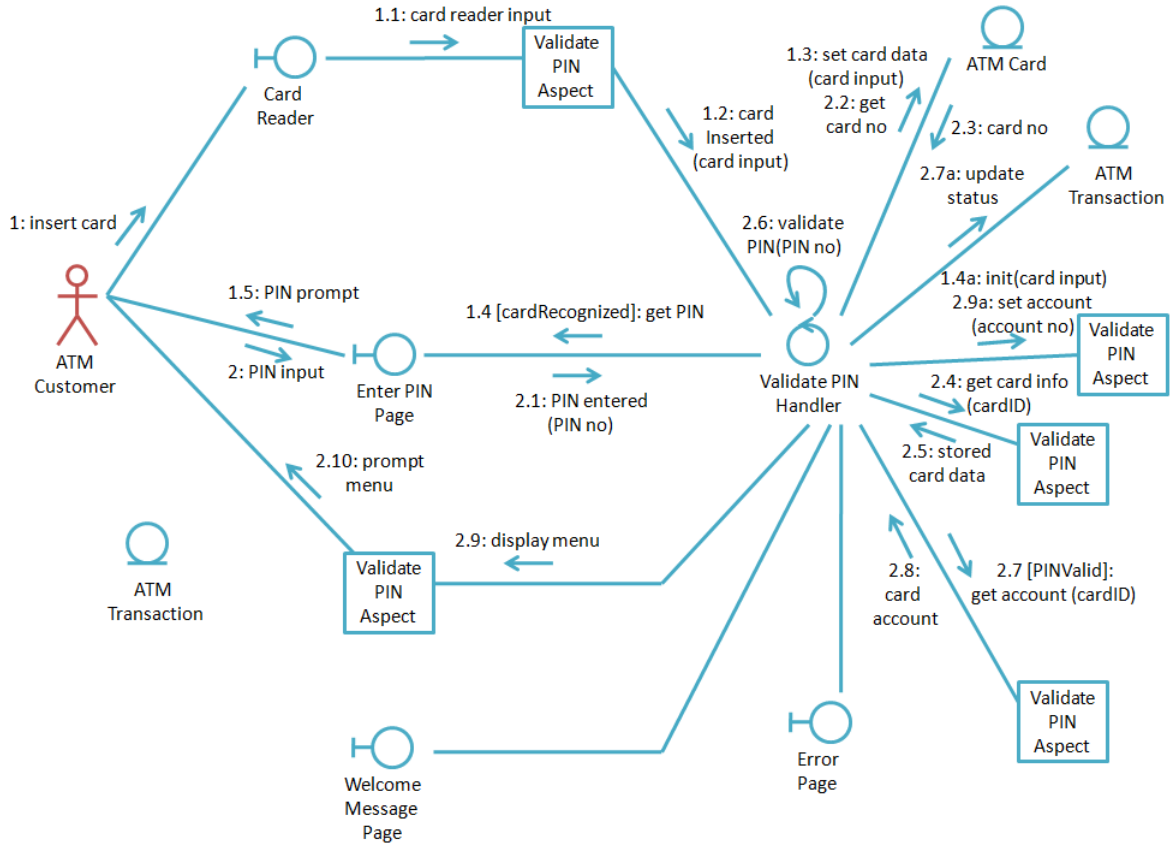

Figure 5.6: Example of AOSD Analysis Model - Collaboration Diagram for *Validate PIN* Use Case Typical Flow

Design Model

In both UDSD and AOSD design phase, the design model is created based on the analysis model. The analysis model has been considered in more detail; we consider more implementation constraints, reorganize the name of classes, aspects, and methods to live up to the standard of Java and AspectJ, consider the distributed issue of the ATM system. The ATM system is a client-server system. Therefore, we divide the system into client side and server side.

In UDSD, the class diagram and collaboration diagram for each use case are refined in order to meet the implementation constraints. And in AOSD, the use-case slice and collaboration diagram for each use case are also refined. The process of design phase is the same as in analysis phase.

5.1.4 System after Change

One of our metrics is change impact metric. For measuring this metric, we need to apply some changes to the system to realize how change affects the system. In our case study, ATM system, we make new requirement by adding new use case to the system;

- *Borrow Money* use case. This use case describes an event that a customer cannot withdraw specific amount of funds from a bank account because of insufficient amount of money in the bank account, so the customer requests to borrow the money from the bank. This use case is an extension use case of the *Withdraw Funds* use case. The detail of this use case is described in Appendix A.

After applying this change, the new system use case model is shown in Figure 5.7.

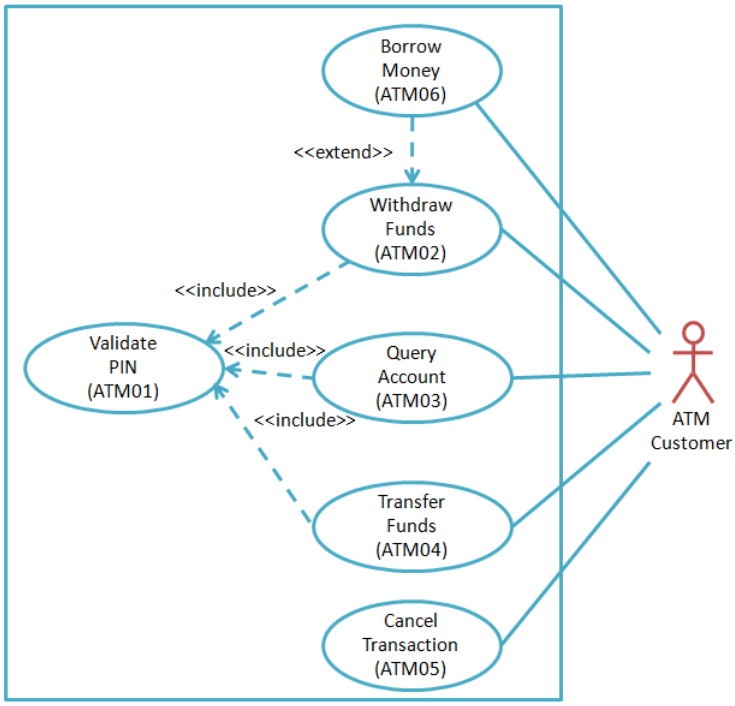

Figure 5.7: ATM System Use Case Diagram after Applying Change

The process after we add *the Borrow Money* use case to the system is the same way as the process before applying change. In UDSD, we create class diagram and collaboration diagram for this use case and refine the former artifacts. In AOSD, we create use-case slice and collaboration diagram for this use case and refine artifacts we created before the change.

5.2 Results of Measurement

In the step of creating the system to be measured, we create the ATM system from requirements phase to design phase for both UDSD and AOSD and add new requirement to the system in order that the change impact metrics can be applied. This section presents the results of the measurement for each metric on the ATM system.

5.2.1 Measures of Change Impact Metrics

In order to measure the change impact metrics suite, we created the ATM system and then added the *Borrow Money* use case as a change to the system. We measured the metric from each artifacts (diagrams) created during the development process by collecting the number of components and relationships in each diagram and calculated the degree of change impact metric for each diagram. Moreover, in order to measure the overall impact of the change, we collected components and relationships in every diagram and calculated the metric. The results of the measurement of the degree of change impact metric I are shown in Table 5.1. In addition to the values I, we also show the detail of the number of components and relationships which are affected by the change (*Imp(c)+Imp(r)*) and the number of components and relationships of the entire system (*Sys(c)+Sys(r)*).

Table 5.1: Measures of Change Impact Metric

Phase	**Diagram**	**UDSD**			**AOSD**		
		Imp(c)+ Imp(r)	*Sys(c)+ Sys(r)*	*I*	*Imp(c)+ Imp(r)*	*Sys(c)+ Sys(r)*	*I*
Requirements	Use Case Diagram	3	10	0.300	3	10	**0.300**
Analysis	Class Diagram/ Use-Case Slice	26	76	0.342	24	86	**0.279**
	Collaboration Diagram	33	116	0.284	30	121	**0.248**
Design	Class Diagram/ Use-Case Slice	32	98	0.327	30	117	**0.256**
	Collaboration Diagram	40	141	0.284	38	151	**0.252**
Overall		134	441	0.304	125	485	**0.258**

Since, the raw data cannot be interpreted without statistical analysis, we will analyze these data statistically in Section 5.3.2. Then, the interpretation of the data will be described in Section 5.4.1.

5.2.2 Measures of Scattering, Tangling and Crosscutting Metrics

In order to measure scattering, tangling and crosscutting metrics, we use the class diagrams in UDSD and the use-case slices in AOSD from analysis and design phase as materials to trace the dependency between use cases and modules (classes or aspects). First, we consolidate class diagrams from all use cases into consolidated class diagram for UDSD system, and also consolidated use-case slice for AOSD system. Then, we create the dependency matrix for the system and calculate the crosscutting product matrix and crosscutting matrix. Note that, the process of creating these matrices are described in Section 4.2.2. Next, we calculate all the metrics from the dependency matrix, crosscutting product matrix and crosscutting matrix.

For the scattering (*NScattering* and *Degree of scattering*) and crosscutting (*Crosscut-points*, *NCrosscut*, and *Degree of Crosscutting*) metrics, we calculate each of them for each use case. In our case study, the system consists of five use cases; *ATM01 Validate PIN*, *ATM02 Withdraw Funds*, *ATM03 Query Account*, *ATM04 Transfer Funds*, and *ATM05 Cancel Transaction*. Then, we calculate the average of these values. Moreover, we calculate these metrics for analysis phase and design phase. The results of measurement are shown in Table 5.2 and Table 5.3.

Table 5.2: Measures of Scattering and Crosscutting Metric of UDSD System

		UDSD									
		Analysis					**Design**				
		NScattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting	NScattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting
Use Case	*ATM01*	11	0.48	11	4	0.54	14	0.48	14	4	0.53
	ATM02	15	0.65	15	4	0.68	18	0.62	18	4	0.65
	ATM03	9	0.39	9	4	0.46	12	0.41	12	4	0.47
	ATM04	11	0.48	11	4	0.54	14	0.48	14	4	0.53
	ATM05	6	0.26	6	4	0.36	6	0.21	6	4	0.29
Global/Average			0.45			0.52		0.44			0.49

Table 5.3: Measures of Scattering and Crosscutting Metric of AOSD System

		AOSD									
		Analysis					Design				
		NScattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting	NScattering	Degree of scattering	Crosscutpoints	NCrosscut	Degree of crosscutting
Use Case	*ATM01*	10	0.37	10	4	0.44	14	0.38	14	4	0.43
	ATM02	13	0.48	13	4	0.53	17	0.46	17	4	0.50
	ATM03	8	0.30	8	4	0.38	12	0.32	12	4	0.38
	ATM04	10	0.37	10	4	0.44	14	0.38	14	4	0.43
	ATM05	6	0.22	6	4	0.31	6	0.16	6	4	0.24
Global/Average			0.35			0.42		0.34			0.39

For the tangling (*NTangling*, *Degree of tangling*) metrics, we calculate these metrics for each module (class or aspect). Then, we calculate the *GTangling* metric which is the average of *Degree of tangling*. The values of *GTangling* for both UDSD and AOSD in analysis and design phase are shown in Table 5.4.

Table 5.4: Measures of Tangling Metric

Metric	**UDSD**		**AOSD**	
	Analysis	**Design**	**Analysis**	**Design**
GTangling	0.34	0.32	0.21	0.20

Similarly to the results of change impact metrics suite, the raw data cannot be interpreted without statistical analysis, we will analyze the data of scattering, tangling, and crosscutting metrics suite statistically in Section 5.3.2. Then, the interpretation of the data will be described in Section 5.4.2.

5.3 Statistical Analysis for Our Results Using T-Test

Although, the results in section 5.2.1 and 5.2.2 show that AOSD can help improve the maintainability and reduce the effect of crosscutting concern problems, consisting of scattering and tangling, in UDSD, but the difference of the values of these metrics are quite small. The difference of values of each metric of UDSD and AOSD is shown in Table 5.5.

Table 5.5: Difference of Measures between UDSD and AOSD

Metric	UDSD		AOSD		Difference (UDSD - AOSD)	
	Analysis	**Design**	**Analysis**	**Design**	**Analysis**	**Design**
Degree of Change impact I of class diagram/ use-case slice	0.342	0.327	0.279	0.256	0.063	0.071
Degree of Change impact I of collaboration diagram	0.284	0.284	0.248	0.252	0.036	0.032
GScattering	0.452	0.441	0.348	0.341	0.104	0.100
GTangling	0.339	0.324	0.215	0.200	0.124	0.124
Average of *Degree of crosscutting*	0.514	0.494	0.419	0.395	0.095	0.099

However, we cannot conclude that AOSD approach is more effective than UDSD approach by just the subtraction of two sets of results. Therefore, we have to find some evidence showing that the differences are significant. In our research, we applied t-test in order to prove that our results have statistical significance and be able to conclude our results properly.

5.3.1 T-Test Definition and Procedure

The t-test is probably the most commonly used statistical data analysis procedure for hypothesis testing. Actually, there are several kinds of t-tests, but the most common is the "two-sample t-test" or also known as the "Student's t-tes". The two sample t-test simply tests whether or not the two independent populations have different mean values on some measure [21, 22].

For example, in our research, we have the hypothesis that our results from AOSD and from UDSD have the significant difference. Therefore, our null hypothesis, which is assumed to be true until proven wrong, is that there is really no difference between the results from UDSD and from AOSD.

In our research, we gather the results from UDSD and AOSD system and observe that the results from AOSD system are lower than the results from UDSD system. However, we have to find the evidence that the differences represent whether the real difference between the two populations, or just a chance difference in our samples.

The statistics t-test allows us to answer this question by using the t-test statistic to

determine a p-value that indicates how likely we could have gotten these results by chance. By convention, if there is less than 5 percent chance of getting the observed differences by chance, we reject the null hypothesis and conclude that we found a statistically significant difference between the two groups.

The procedure of t-test is as follows;

1. Set the null hypothesis that the means of two groups are the same. And the alternative hypothesis is that the means of two groups are different.

2. Compute the values needed for continuing t-test; the mean of group one $\overline{X}_1$, the mean of group two $\overline{X}_2$, standard deviation of group one S_1, standard deviation of group two S_2, the sample size in group one n_1, the sample size in group two n_2, the degree of freedom $d.f.$, the estimator of the standard deviation of the two groups $S_{\overline{X}_1-\overline{X}_2}$, and the t statistic value. The formulas for $d.f.$, $S_{\overline{X}_1-\overline{X}_2}$, t are shown in Table 5.6. Note that, there are different formulas for the two groups that have equal variance and unequal variance.

Table 5.6: T-Test Formulas

Variance of Two Groups	$\boldsymbol{d.f.}$	$\boldsymbol{S_{\overline{X}_1-\overline{X}_2}}$	$\boldsymbol{t}$
Equal	n_1+n_2-2	$\sqrt{\dfrac{(n_1-1)S_1^2+(n_2-1)S_2^2}{(n_1+n_2-2}}$	$\dfrac{\overline{X}_1-\overline{X}_2}{S_{\overline{X}_1-\overline{X}_2}\sqrt{\dfrac{1}{n_1}+\dfrac{1}{n_2}}}$
Unequal	$\dfrac{\left(\dfrac{S_1^2}{n_1}+\dfrac{S_2^2}{n_2}\right)^2}{\dfrac{\left(\dfrac{S_1^2}{n_1}\right)^2}{n_1-1}+\dfrac{\left(\dfrac{S_2^2}{n_2}\right)^2}{n_2-1}}$	$\sqrt{\dfrac{S_1^2}{n_1}+\dfrac{S_2^2}{n_2}}$	$\dfrac{\overline{X}_1-\overline{X}_2}{S_{\overline{X}_1-\overline{X}_2}}$

3. Calculate the p-value with the specified degree of freedom by looking into the t-distribution table. The p-value is the probability value of a t-test. If the p-value is less than 0.05, this means that there is less than 5 percent chance of getting the observed differences by chance, we reject the null hypothesis and conclude that we found a statistically significant difference between the two groups.

5.3.2 T-Test Calculation for Our Metrics Results

In our research, we measured each of our metrics from the system implemented by UDSD and the system implemented by AOSD. The results show that the measures from AOSD are lower than UDSD, but the difference seems to be small. We have to find the evidence whether the difference is significant or not. Therefore, the t-test has been organized. In this section, we show the results of t-test calculation for each metric. Note that, we cannot assume that the variance of UDSD system and AOSD system is equal or unequal, so we calculate the t-test for both cases.

T-Test Calculation for Change Impact Metrics

In our research, we measured Degree of Change Impact I metric for each diagram created in the development process; use case diagram, analysis class diagram, analysis collaboration diagram, design class diagram, and design collaboration diagram for UDSD system, and use case diagram, analysis use-case slice, analysis collaboration diagram, design use-case slice, and design collaboration diagram for AOSD system. However, the Degree of Change Impact I measures for both UDSD and AOSD are the same value because we created the same use case diagram for both approaches. Therefore, we do not include the Degree of Change Impact I measures to the t-test calculation. The results of t-test calculation for Degree of Change Impact I metric are shown in Table 5.7. The measures from UDSD system is defined as group one and the measure from AOSD system is defined as group two.

Table 5.7: T-Test Calculation for Degree of Change Impact I Measures

	Variable	**UDSD (Group 1)**	**AOSD (Group 2)**
	$\overline{X}$	0.3093	0.2588
	n	4	4
	S	0.0298	0.0139
Equal Variance	$d.f.$	6	
	$S_{\overline{X}_1-\overline{X}_2}$	3.072	
	t	3.072	
	p-value	**0.0109**	
Unequal Variance	$d.f.$	4.246	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0164	
	t	3.079	
	p-value	**0.0185**	

From the t-test calculation result, we can notice that the p-value for equal variance case is 0.0109 and the p-value for unequal variance case is 0.0185 which they are lower than 0.05. Therefore, we can reject the null hypothesis and conclude that the difference

between the average of Degree of Change Impact I of UDSD system and AOSD system is statistically significant.

T-Test Calculation for Scattering Metrics

For the scattering metrics, we use *GScattering* metric to compare UDSD system and AOSD system. The *GScattering* metric is the average of the values of *Degree of scattering* metric of each use case. Therefore, we use the values of *Degree of scattering* metric as the data in the group and the value of *GScattering* metric as the mean of the group. In our research, we collected the values of *Degree of scattering* metric in analysis and design phase separately. Therefore, we organize the t-test for the results of both phases. The results of t-test calculation for scattering metric values at analysis phase and design phase are shown in Table 5.8 and Table 5.9, respectively.

Table 5.8: T-Test Calculation for Scattering Metric Measures at Analysis

	Variable	**UDSD (Group 1)**	**AOSD (Group 2)**
	$\overline{X}$	0.4522	0.3481
	n	5	5
	S	0.1429	0.09669
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1220	
	t	1.350	
	p-value	**0.1070**	
Unequal Variance	$d.f.$	7.024	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0771	
	t	1.350	
	p-value	**0.1095**	

Table 5.9: T-Test Calculation for Scattering Metric Measures at Analysis

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.4414	0.3405
	n	5	5
	S	0.1511	0.1108
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1325	
	t	1.204	
	p-value	**0.1315**	
Unequal Variance	$d.f.$	7.337	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0838	
	t	1.204	
	p-value	**0.1339**	

From the t-test calculation result, we can notice that the p-values for scattering metrics of both analysis and design phase are higher than 0.05. Therefore, we can conclude that the null hypothesis is true and the difference between the values of *GScattering* of UDSD system and AOSD system is not statistical significant.

T-Test Calculation for Tangling Metrics

Similarly to scattering metrics, for the tangling metrics, we use *GTangling* metric to compare UDSD system and AOSD system. The *GTangling* metric is the average of the values of *Degree of tangling* metric of each module in the system. The module means class or aspect. Therefore, we use the values of *Degree of tangling* metric as the data in the group and the value of *GTangling* metric as the mean of the group. In our research, we collected the values of *Degree of tangling* metric in analysis and design phase separately. Therefore, we organize the t-test for the results of both phases. The results of t-test calculation for tangling metric values at analysis phase and design phase are shown in Table 5.10 and Table 5.11, respectively.

Table 5.10: T-Test Calculation for Tangling Metric Measures at Analysis

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.3391	0.2148
	n	23	27
	S	0.4197	0.3840
Equal Variance	$d.f.$	48	
	$S_{\overline{X}_1-\overline{X}_2}$	0.4008	
	t	1.093	
	p-value	**0.1399**	
Unequal Variance	$d.f.$	45.14	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1145	
	t	1.085	
	p-value	**0.1419**	

Table 5.11: T-Test Calculation for Tangling Metric Measures at Design

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.3241	0.200
	n	29	37
	S	0.4120	0.3682
Equal Variance	$d.f.$	64	
	$S_{\overline{X}_1-\overline{X}_2}$	0.3880	
	t	1.290	
	p-value	**0.1008**	
Unequal Variance	$d.f.$	56.74	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0976	
	t	1.272	
	p-value	**0.1043**	

From the t-test calculation result, we can notice that the p-values for tangling metrics of both analysis and design phase are higher than 0.05. Therefore, we can conclude that the difference between the values of *GTangling* of UDSD system and AOSD system is not statistical significant.

T-Test Calculation for Crosscutting Metrics

For the crosscutting metrics, we use the average of the *Degree of crosscutting* metric of each use case to compare UDSD system and AOSD system. Therefore, we use the values of *Degree of crosscutting* metric as the data in the group. In our research, we collected the

values of *Degree of crosscutting* metric in analysis and design phase separately. Therefore, we organize the t-test for the results of both phases. The results of t-test calculation for crosscutting metric values at analysis phase and design phase are shown in Table 5.12 and Table 5.13, respectively.

Table 5.12: T-Test Calculation for Crosscutting Metric Measures at Analysis

	Variable	**UDSD (Group 1)**	**AOSD (Group 2)**
	$\overline{X}$	0.5143	0.4188
	n	5	5
	S	0.1174	0.0815
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1011	
	t	1.494	
	p-value	**0.0868**	
Unequal Variance	$d.f.$	7.129	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0639	
	t	1.494	
	p-value	**0.0894**	

Table 5.13: T-Test Calculation for Crosscutting Metric Measures at Design

	Variable	**UDSD (Group 1)**	**AOSD (Group 2)**
	$\overline{X}$	0.4941	0.3952
	n	5	5
	S	0.1289	0.0976
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1143	
	t	1.368	
	p-value	**0.1043**	
Unequal Variance	$d.f.$	7.452	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0723	
	t	1.368	
	p-value	**0.1068**	

From the t-test calculation result, we can notice that the p-values for crosscutting metrics of both analysis and design phase are higher than 0.05. Therefore, we can conclude that the difference between the average values of *Degree of crosscutting* metric of UDSD system and AOSD system is not statistical significant.

5.4 Discussion

After the measurements and the calculation of t-test for each metrics suite, we can discuss about our results based on the data from measurements and the results of t-test. First, we discuss about the results from change impact metrics suite. Then, we discuss about the results from scattering, tangling, and crosscutting metrics suite.

5.4.1 Discussion for Change Impact Metrics

According to the definition of change impact metric described in Section 4.2, we can compare UDSD complexity and AOSD complexity in terms of effect of change by comparing the values I measured from the diagrams in the same level of abstraction.

In the requirements phase, we can compare the use case diagram of the two approaches. However, both UDSD and AOSD use the common use case diagram, so there is no difference between these two values.

In the analysis and design phase, the measures of change impact metric from UDSD class diagram and AOSD use-case slice are compared. Moreover, the measures from collaboration diagrams from both approaches are also compared. From the values shown in Table 5.1, we can notice that the measure values *I* of UDSD are higher than the measure values *I* of AOSD for all diagrams. This can directly refer that when a change occurs, the effect of change to the system implemented by UDSD is more than the effect of change to the system implemented by AOSD. Then, we look at this comparison in more detail. Comparing to UDSD, the lower I values of AOSD come from the lower *Imp(c) + Imp(r)* and the higher *Sys(c) + Sys(r)* as shown in Figure 5.8. This means that the number of components and relationships of AOSD that are affected by the change is lower than those in UDSD and the number of components and relationships of the AOSD entire system is higher than those in UDSD entire system.

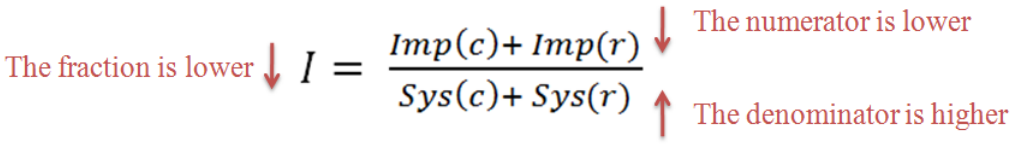

Figure 5.8: Explanation of Lower Change Impact in AOSD Comparing to UDSD

For the overall degree of change impact, we calculated from the sum of the number of components and relationships affected by the change from all diagrams divided by the number of components and relationships from all diagrams. According to the results shown in Table 5.1, the measure of degree of change impact for AOSD is lower than the measure for UDSD. These measure values are in the same trend as the comparison of measures from each diagram.

One of our objectives is to evaluate how AOSD improve maintainability in UDSD. The change impact metric can be used to refer to the maintainability of the system. The more

effects the system receives from change, the less ease to maintain of the system is. From this case study, we can notice that the system implemented by UDSD received more effect from change than the system implemented by AOSD. Therefore, we can infer that the AOSD system is easier to maintain than UDSD system.

According to the results of t-test, we can conclude that our results for degree of change impact I metric can be used to identify the difference between the ATM systems implemented by UDSD and AOSD in term of maintainability. In this case, the AOSD can help increase maintainability on UDSD according to the raw data of measurements and the difference is significant according to the statistic analysis.

5.4.2 Discussion for Scattering, Tangling, and Crosscutting Metrics

One of our research objectives is to evaluate how much AOSD reduce the crosscutting concerns problem in UDSD. The scattering, tangling, and crosscutting metrics can derive this attribute of the software system. Since, crosscutting concerns can be divided into two problems; scattering and tangling.

For the scattering metrics, in both analysis and design phase, please consider the GScattering values which are the average values of Degree of scattering metric of each use case. The GScattering values of AOSD are lower than those of UDSD. This means AOSD modules (classes and aspects) are less scattered than UDSD modules (classes). This is because the unique characteristic of AOSD that it uses aspects to encapsulate the part of classes (methods of classes) that are specific to the use case. Therefore, it reduces the number of modules of the use case and the modules that fulfill a specified use case are less spread to the system than modules fulfilling the same use case in UDSD system.

For the tangling metrics, please consider the values of GTangling metric in both analysis and design phase. The GTangling metric is a metric that is calculated from the average values of the Degree of tangling metric of each module of the system. According to the results, the GTangling values of AOSD are lower than those of UDSD. This means the modules in AOSD system are less tangled with many use cases than the modules in UDSD system. This is because the advantage of aspects that they can encapsulate parts of the classes specific to use case in one containing module. Therefore, aspects reduce effect of tangling problem in classes.

For the crosscutting metrics, please consider the values of the average of Degree of crosscutting metric in both analysis and design phase. The Degree of crosscutting metric is the combination of scattering and tangling. It considers how much modules of specified use case is spread to the system by calculating *Crosscutpoints* metric and considers how much this use case cut across other use cases in the system by calculating *NCrosscut* metric. According to the results, the values of the average of Degree of crosscutting metric of AOSD system are lower than those of UDSD system. This means, in the AOSD system, the use cases cut across each other and the modules of use case are spread to the system less than in UDSD system.

However, according to the t-test results for scattering, tangling, and crosscutting met-

rics, our results of these metrics cannot be used to identify the difference between the ATM systems implemented by UDSD and AOSD in term of separation of concerns. The ATM system implemented by AOSD might have less effect of crosscutting concerns than the ATM system implemented by UDSD according to the raw data but the efficiency of AOSD is too low according to the statistic analysis.

5.5 Effect of AOSD Characteristic on Our Results

AOSD is said to be an approach that can help increase the maintainability and reduce the effect of crosscutting concerns of the UDSD approach. In term of maintainability, AOSD uses aspects to encapsulate new components and new relationships that fulfill the change and reduce the effect of change to the existing system. In term of crosscutting concerns, AOSD uses aspects to encapsulate modules which are specific to the use case in order to keep use cases separate from each other

According to the advantages of using aspects, we expected that AOSD can increase the efficiency of UDSD dramatically. However, our results show that AOSD might help increase the maintainability and separation of concerns of UDSD but not in a large scale. In this section, we describe why the difference of results from both approaches is quite small.

5.5.1 The Ideal Case and Practical Case for Crosscutting Concerns

The situation of crosscutting concerns is described as some concerns in the system affect more than one component and some components contain parts of multiple concerns. This is the combination of scattering and tangling, respectively. For simplicity, please look at the Hotel Management System which we used as an example to describe scattering and tangling problems in Chapter 2. In Figure 5.9, we show that after realizing each use case in the system, some components contain parts that fulfill different use cases and ideally in a component that contains several parts, these parts are not related. In this case, each part contains methods and attributes which are implemented to fulfill a certain use case. For example, the *Room* component contains three parts that fulfill *Reserve Room* use case, *Check In Customer* use case, and *Check Out Customer* use case but these three parts are not related.

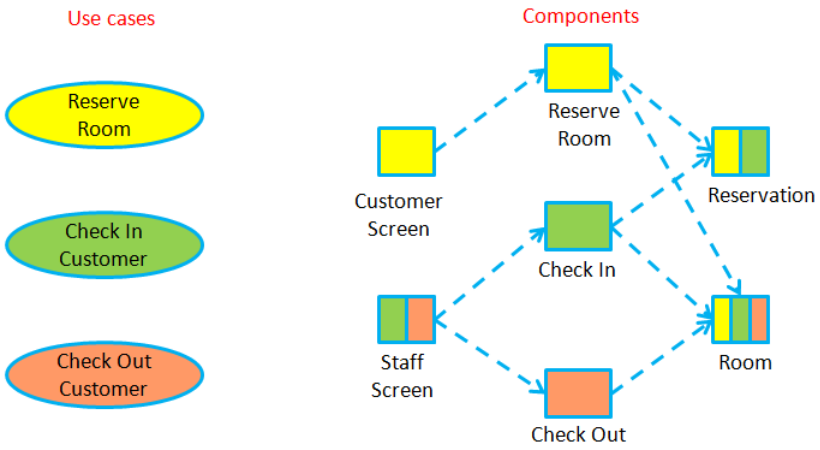

Figure 5.9: The Ideal Case for Crosscutting Concerns

For this crosscutting concerns situation, AOSD uses aspects to encapsulate the parts that are specific to a certain use case and then it uses use-case slice to encapsulate the classes that are specific to the use case and the aspect. For example, in Figure 5.10, *Reserve Room* use-case slice contains *Customer Screen* class and *Reserve Room* class which are used only for this use case and part of *Reservation* class and part of *Room* class are put in the *Reserve Room* aspect as one containing module. As a result, the *Reserve Room* use case does not have to depend on the base [1] *Reservation* class and the base *Room* class but it depends on *Reserve Room* aspect instead. Note that, in this case, the base class means a class that is used to fulfill a certain use case but it is also used by other use cases. Therefore, it is divided into several parts to fulfill different use cases but because of the use of aspects to encapsulate parts of the base classes which are specific to the use case, the use case has no dependency to the base class but to the aspect instead. In Figure 5.10, the base classes (*Staff Screen* class, *Reservation* class, and *Room* class) are shown in the lowest row.

[1]The base class means a class that is used to fulfill a certain use case but it is also used by other use cases. After applying aspects to encapsulate the parts that are specific to a certain use case, there are still base classes for aspects to refer and merge the specific parts together.

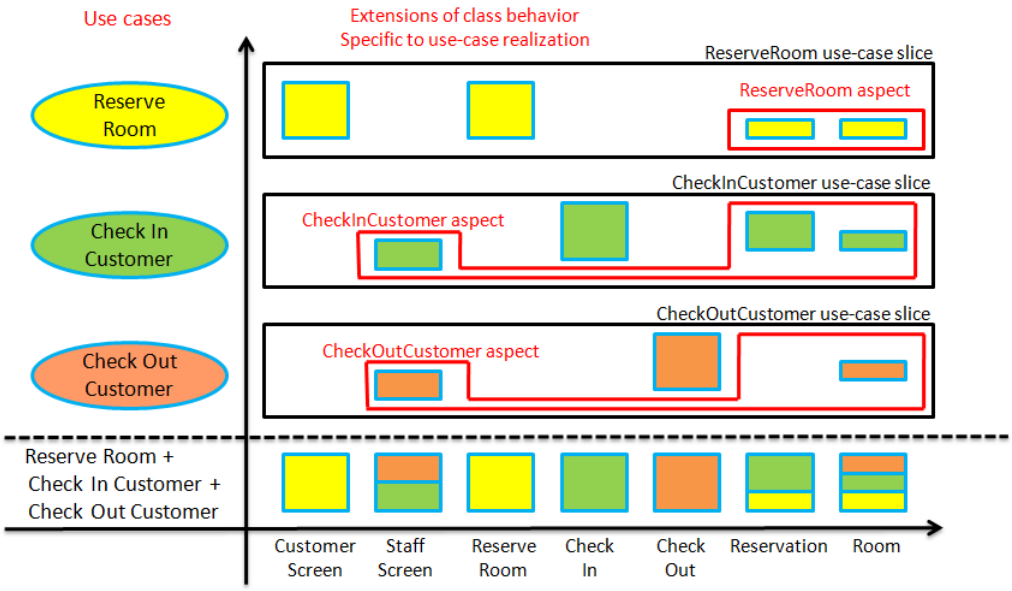

Figure 5.10: Use-Case Slice and Aspect for Ideal Crosscutting Concerns

For this ideal case, the results of the measurement of our metrics would show a big difference between UDSD system and AOSD system. Since, for the change impact metrics, when change occurs, we realize the change requirements and encapsulate them with aspects and use-case slices. Therefore, the change will not affect the base classes and as a result, the number of components and relationships in the modified part (*Mod(c)+Mod(r)*), that is affected by the change, in the AOSD system will decrease. Then, AOSD system receives less effect from change than UDSD system.

For the scattering, tangling, and crosscutting metrics, there is no more dependency between use case and base classes, but there is a dependency between use case and an aspect instead. As a result, the number of modules which fulfill a certain use case will decrease, so the effect of scattering is lower in AOSD system. Moreover, there is no more tangling in the AOSD system because parts that are specific to a certain use case are encapsulated in an aspect. Finally, there is no more crosscutting between use cases.

However, in the real world, for a component that contains several parts, these parts have some methods or attributes that can be used in many use cases. For simplicity, we call these parts as "common parts". The practical case for crosscutting concerns is shown in Figure 5.11. For example, in the *Room* class, there is a method called *retrieve()* to retrieve data of a room. This method is used in common for *Reserve Room* use case, *Check In Customer* use case, and *Check Out Customer* use case. Therefore, it is put in the common parts.

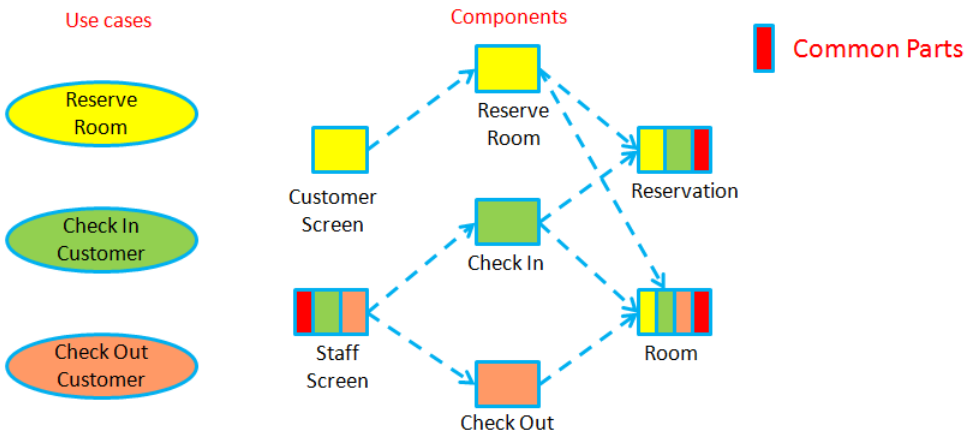

Figure 5.11: The Practical Case for Crosscutting Concerns

5.5.2 The Use of Non-Use-Case-Specific Slice

In order to manage the common parts, AOSD provides non-use-case-specific slice as containment for them. When we realize use cases, if there are methods that are used in many use cases, we put these methods in non-use-case-specific slice. For example, in ATM system, there are four non-use-case-specific slices for containing the common parts of classes; *I/O Handler* slice, *Customer Interface* slice, *Bank Data Management* slice, and *ATM Data Management* slice. When a use-case slice needs to use these common parts, we just put the extend dependency to these non-use-case-specific slices. In Figure 5.12, we show *Validate PIN* use-case slice extending the four non-use-case-specific slices.

As a consequence of using non-use-case-specific slices to contain the common parts, for scattering, tangling, and crosscutting metrics, the use case still has the dependency to the base classes and also has a dependency to an aspect. Consequently, use cases are more scattered than the ideal case of crosscutting concerns. Some classes that have common parts are still tangled because use cases use the common parts, and then there still has dependency relationship between use case and class. Moreover, use cases cut across each other because they use the same methods in common parts.

For the change impact metrics, some classes still have dependency to the base classes to access the common parts' methods and also have dependency to an aspect. Therefore, when change occurs, the change can affect the base classes, and then the number of components and relationships in modified part (Mod(c)+Mod(r)) will increase from the ideal case of crosscutting concerns.

To sum up, the use of non-use-case-specific slice to contain the common parts hinders the efficiency of AOSD on the improvement of maintainability and the reduction of crosscutting concerns of UDSD. Therefore, we can notice from our results that the difference of results between UDSD system and AOSD system are quite small.

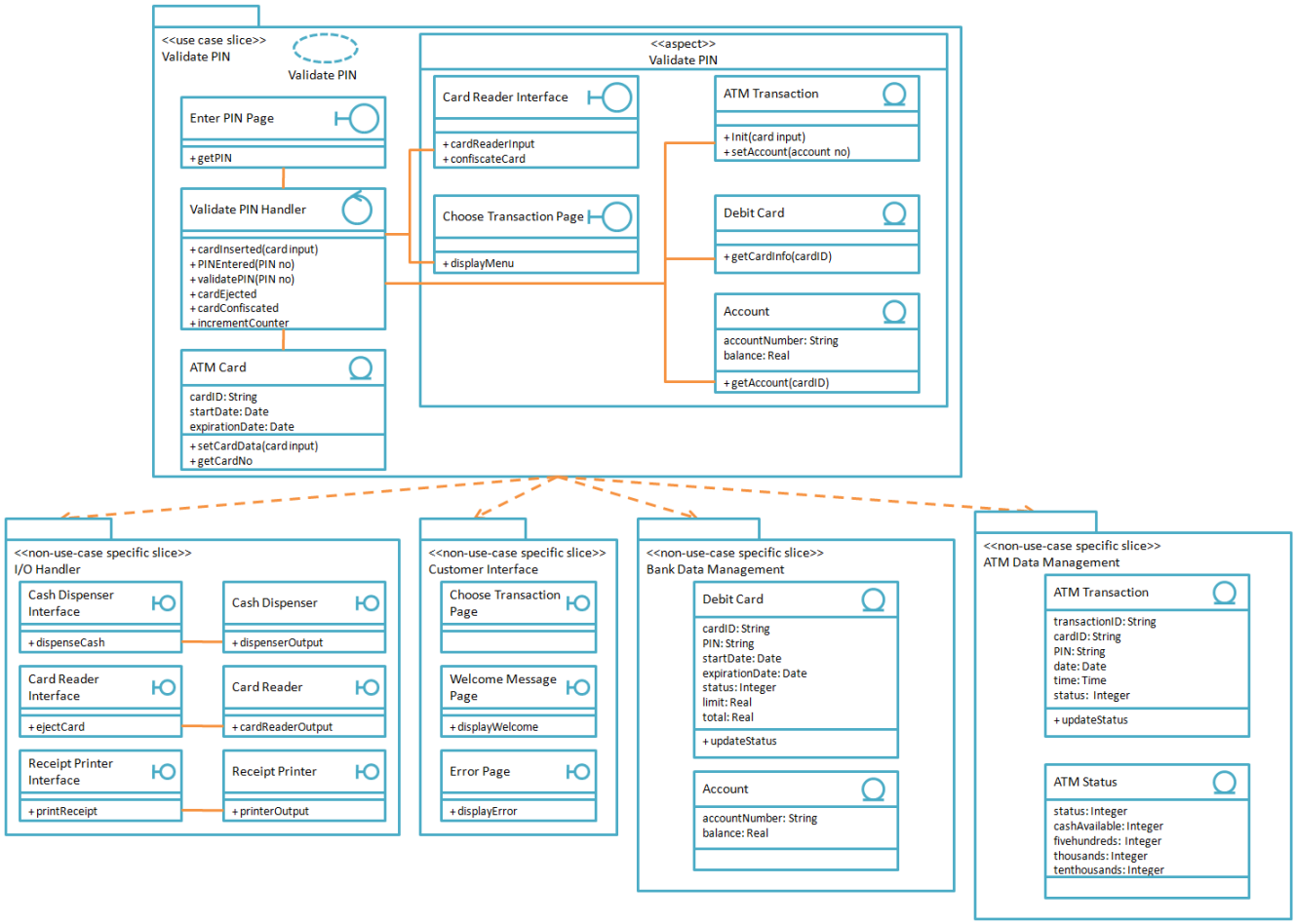

Figure 5.12: *Validate PIN* Use-Case Slice Extending the Four Non-Use-Case-Specific Slices

5.5.3 AOSD System without Non-Use-Case-Specific Slice

Let's imagine that if we ignore the use of non-use-case-specific slices, we have to duplicate the common methods and put the common parts together with specific parts of the classes in the aspect in each use-case slice. The results measured from the ATM system implemented by AOSD without non-use-case-specific slice (NUCS) are shown in Table 5.14. We also calculate the difference between the results of ATM system implemented by UDSD and the results of ATM system implemented by AOSD without non-use-case-specific slice.

Table 5.14: Results of ATM System Implemented by AOSD without NUCS

Metric	**UDSD**		**AOSD without NUCS**		**Difference (UDSD - AOSD)**	
	Analysis	**Design**	**Analysis**	**Design**	**Analysis**	**Design**
Degree of Change impact *I* of class diagram/ use-case slice	0.342	0.327	0.123	0.125	0.219	0.202
Degree of Change impact *I* of collaboration diagram	0.284	0.284	0.169	0.170	0.115	0.114
GScattering	0.452	0.441	0.119	0.135	0.333	0.306
GTangling	0.339	0.324	0.000	0.000	0.339	0.324
Average of *Degree of crosscutting*	0.514	0.494	0.100	0.119	0.414	0.375

Again, we cannot conclude the difference of results with only the subtraction of results. Therefore, we do the t-test to prove whether the difference of our results has statistic significance or not.

T-Test Calculation for Change Impact Metrics

The result of t-test for change impact I measures in case of UDSD and AOSD without non-use-case-specific slice is shown in Table 5.15.

From the t-test calculation result, we can notice that the p-value for equal variance case and the p-value for unequal variance case is 0.00009 which they are lower than 0.05. Therefore, we can reject the null hypothesis and conclude that the difference between the average of Degree of Change Impact I of UDSD system and AOSD system without non-use-case-specific slice is statistically significant.

Table 5.15: T-Test Calculation for Degree of Change Impact I Measures (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.3093	0.1468
	n	4	4
	S	0.0298	0.0263
Equal Variance	$d.f.$	6	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0281	
	t	8.177	
	p-value	**0.00009**	
Unequal Variance	$d.f.$	5.909	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0199	
	t	8.177	
	p-value	**0.00009**	

T-Test Calculation for Scattering Metrics

The results of t-test calculation for scattering metric values at analysis and design phase in case of UDSD and AOSD without non-use-case-specific slice are shown in Table 5.16 and Table 5.17, respectively.

Table 5.16: T-Test Calculation for Scattering Metric Measures at Analysis (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.4522	0.1185
	n	5	5
	S	0.1429	0.0609
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1098	
	t	4.804	
	p-value	**0.00067**	
Unequal Variance	$d.f.$	5.407	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0695	
	t	4.804	
	p-value	**0.00243**	

Table 5.17: T-Test Calculation for Scattering Metric Measures at Design (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.4414	0.1351
	n	5	5
	S	0.1511	0.0604
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.1151	
	t	4.209	
	p-value	**0.00148**	
Unequal Variance	$d.f.$	5.246	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0728	
	t	4.209	
	p-value	**0.00421**	

From the t-test calculation result, we can notice that the p-values for scattering metrics of both analysis and design phase are lower than 0.05. Therefore, we can reject the null hypothesis and conclude that the difference between the values of *GScattering* of UDSD system and AOSD system without non-use-case-specific slice is statistical significant.

T-Test Calculation for Tangling Metrics

The results of t-test calculation for tangling metric values at analysis and design phase in case of UDSD and AOSD without non-use-case-specific slice are shown in Table 5.18 and Table 5.19, respectively.

Table 5.18: T-Test Calculation for Tangling Metric Measures at Analysis (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.3391	0
	n	23	27
	S	0.4197	0
Equal Variance	$d.f.$	48	
	$S_{\overline{X}_1-\overline{X}_2}$	0.2841	
	t	4.206	
	p-value	**0.00006**	
Unequal Variance	$d.f.$	22	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0875	
	t	3.875	
	p-value	**0.00041**	

Table 5.19: T-Test Calculation for Tangling Metric Measures at Design (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.3241	0
	n	29	37
	S	0.4120	0
Equal Variance	$d.f.$	64	
	$S_{\overline{X}_1-\overline{X}_2}$	0.2725	
	t	4.795	
	p-value	**0.000005**	
Unequal Variance	$d.f.$	28	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0765	
	t	4.236	
	p-value	**0.00011**	

From the t-test calculation result, we can notice that the p-values for tangling metrics

of both analysis and design phase are lower than 0.05. Therefore, we can reject the null hypothesis and conclude that the difference between the values of *GTangling* of UDSD system and AOSD system without non-use-case-specific slice is statistical significant.

T-Test Calculation for Crosscutting Metrics

The results of t-test calculation for crosscutting metric values at analysis and design phase in case of UDSD and AOSD without non-use-case-specific slice are shown in Table 5.20 and Table 5.21, respectively.

Table 5.20: T-Test Calculation for Crosscutting Metric Measures at Analysis (for UDSD and AOSD without NUCS)

	Variable	**UDSD (Group 1)**	**AOSD (Group 2)**
	$\overline{X}$	0.5143	0.1000
	n	5	5
	S	0.1174	0.0513
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0906	
	t	7.231	
	p-value	**0.00005**	
Unequal Variance	$d.f.$	5.474	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0573	
	t	7.231	
	p-value	**0.00040**	

Table 5.21: T-Test Calculation for Crosscutting Metric Measures at Design (for UDSD and AOSD without NUCS)

	Variable	UDSD (Group 1)	AOSD (Group 2)
	$\overline{X}$	0.4941	0.1190
	n	5	5
	S	0.1289	0.0532
Equal Variance	$d.f.$	8	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0986	
	t	6.015	
	p-value	**0.00016**	
Unequal Variance	$d.f.$	5.324	
	$S_{\overline{X}_1-\overline{X}_2}$	0.0624	
	t	6.015	
	p-value	**0.00091**	

From the t-test calculation result, we can notice that the p-values for crosscutting metrics of both analysis and design phase are lower than 0.05. Therefore, we can reject the null hypothesis and conclude that the difference between the average values of *Degree of crosscutting* metric of UDSD system and AOSD system without non-use-case-specific slice is statistical significant.

To sum up, after ignoring the use of non-use-case-specific slices, we can notice that the results for all of our metrics show the big differences between the ATM system implemented by UDSD and the system implemented by AOSD without non-use-case-specific slices and these differences are statistical significant. Therefore, we can conclude that ignoring the use of non-use-case-specific slices can help the system to become an ideal case of crosscutting concerns and we can reach the full efficiency of AOSD in terms of increasing maintainability and reducing the effect of crosscutting concerns. However, without the non-use-case-specific slices, we need to duplicate the parts which locate in the common parts and put them into the aspect of many use cases. Therefore, it reduces reusability of the system and increases the size of the system. As a result, we have to find the trade-off between the reduction of crosscutting concerns and reusability of the system.

Chapter 6

Conclusion and Future Works

Aspect-Oriented Software Development (AOSD) with use cases proposed by Ivar Jacobson [2] is said to be an approach which helps increase maintainability and reduce the effect of crosscutting concerns of the system implemented by Use Case Driven Software Development (UDSD) [1]. However, there is still no evidence to literally show the efficiency of AOSD over UDSD yet.

In this paper, we proposed one metrics suite called change impact metrics to evaluate how the change affects the system after the change occurs. Since, the maintainability relates directly to the change [17], this metrics suite can refer to the maintainability of the system. It was defined based on the number of components and relationships in each artifact created from requirements phase to design phase. Moreover, we applied one metrics suite called scattering, tangling, and crosscutting metrics suite proposed by Conejero J. et al. to evaluate how much the separation of concerns in the system. This metrics suite was defined based on the traceability dependency between source and target, which in our research, source refers to use case and target refers to module. Then, we derived the dependency matrix, crosscutting product matrix and crosscutting matrix from the traceability dependency relationships. In our research, we used class diagram created in analysis and design phase of UDSD and use-case slice created in analysis and design phase of AOSD as materials to define the dependency between use case and module.

In AOSD, there are some unique features such as aspects, intertype methods and attributes which do not appear in UDSD system. In order to compare the UDSD and AOSD system in a meaningful and consistent manner, we instantiated concern-oriented meta-model proposed by Figueiredo, E. et al. as a base for comparing the two systems [16].

An empirical study was carried out to evaluate our metrics. We used the ATM system which is introduced in "Designing Concurrent, Distributed, and Real-Time Applications with UML" [20] as a case study. We implemented two ATM systems from requirements phase to design phase; one implemented by UDSD and the other implemented by AOSD. Then, in order to measure the change impact metrics, we applied change to the system by introducing new use case. After implementing the two systems, we measured each of our metrics from both systems.

The results of our empirical study show that the AOSD system is more maintainable

and has less effect of crosscutting concerns than the UDSD system. However, there is only small difference between measures of each metric. This is because in the real world some classes contain not only the parts (methods and attributes) that fulfill different use cases and are not related to each other, but also the parts that are used by many use cases, which we called common parts. AOSD provides non-use-case-specific slice to contain these common parts. As a result, use case still has the dependency to the base class in common parts and some classes still has relationships to the base classes, so when the change occurs, it can also affect these common parts. Therefore, the efficiency of AOSD is hindered by the use of non-use-case-specific slices. However, if we remove the use of non-use-case-specific slices out of the system, it reduces the reusability of the system and increases the system size because of the common parts. Consequently, we have to consider the trade-off between separation of concerns and reusability.

Our plans for future works, we concern about the following issues;

- The aspect-oriented software development with use cases proposed by Ivar Jacobson is a holistic approach to develop software system with separation of concerns. He proposed not only the approach for separating the functional concerns from each other, but also the approach for separating nonfunctional concerns from the functional concerns and the approach for separating platform-specific concerns from non-platform-specific concerns. In our research, we just considered the approach for separating the functional concerns from each other, so it is a good plan to explore more about other approaches.

- In our research, the change impact metrics were defined based on the number of components and relationships. In fact, the complexities of all components and relationships are not equal. Therefore, we should consider the complexity of them and refine our change impact metrics.

- The case study that we used in our research was derived from textbook, so it is not the example from the real projects. Therefore, we need to apply our metrics to the projects in the industry to see more different viewpoints. Moreover, the ATM system is just a small project, so we might have more observations by using bigger projects as our case studies.

- The validation of our metrics.

  In software engineering empirical studies, the internal validity and external validity of the studies are one of the concern issues. An internally valid study is one that has been carried out correctly, such that we can have confidence in the study results. In this case, we applied statistical analysis to analyze the confidence of our results statistically, so we can say that our study is internally valid. An external validation requires a study to be confirmed by internal and external replication of its experiments using different new information, so that its finding can be confidently generalized. In the case of external validation, we have to apply our metrics to more case studies in order to generalize our metrics to many different viewpoints.

Bibliography

[1] Ivar Jacobson, Grady Booch, and James Rumbaugh. "The Unified Software Development Process". Addison Wesley Longman Inc, 2000.

[2] Ivar Jacobson, and Pan-Wei Ng. "Aspect-Oriented Software Development with Use Cases". Pearson Education Inc, 2004.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. "Aspect-Oriented Programming". Computer Science Volume 1241/1997, pp. 220-242, 1997.

[4] Tarr, P. et al. "N Derees of Separation: Multi-Dimensional Separation of Concerns". Proceedings of the 21st International Conference on Software Engineering, May 1999.

[5] Horst Zuse. "A Framework of Software Measurement". Walter De Gruyter, 1997.

[6] Everald E. Mills. "Software Metrics". SEI Curriculum Module SEI-CM-12-1.1, December 1998.

[7] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 20(6):476-493, 1994.

[8] Jacqueline A. Mcquillian and James F. Power. "On the Application of Software Metrics to UML Models". MoDELs 2006 Workshops, pp. 217-226, 2007.

[9] Jacqueline A. Mcquillian and James F. Power. "A Definition of the Chidamber and Kemerer Metrics Suite for UML". Technical report, National University of Ireland (2006).

[10] Fenton, N., Lawrence Pfleeger, S. "Software Metrics: A Rigorous and Practical Approach". International Thomson Computer Press, 1996.

[11] M. Ceccato and P. Tonella. "Measuring the Effects of Software Aspectization". In WARE'04 Workshop, 2004.

[12] Kotrappa Sirbi and Prakash Jayanth Kulkarni. "Metrics for Aspect Oriented Programming-An Empirical Study". International Journal of Computer Applications, Vol.5, 2010.

[13] Tarr, P. et al. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. Proceedings of the 21st International Conference on Software Engineering, May 1999.

[14] Sant’fAnna C. et al. “On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework”. SBES’03: Proceedings of the Brazilian Symposium on Software Engineering, pp. 19-34, 2003.

[15] Conejero, J., Figueiredo, E., Garcia, A., Hernandez, J., Jurado, E. “Early Crosscutting Metrics as Predictors of Software Instability”. In 47th International Conference Objects, Models, Components, Patterns (TOOLS), 2009.

[16] Figueiredo, E. et al. “On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework”. Proc. of European Conf. on Soft. Maint. and Reeng. (CSMR). Athens, 2008.

[17] D.M. Coleman, D. Ash, B. Lowther, and P.W. Oman. “Using Metrics to Evaluate Software System Maintainability”. Computer, vol. 27, no. 8, pp. 44-49, Aug. 1994.

[18] Farias, K., Garcia, A. and Whittle, J. “Assessing the Impact of Aspects on Model Composition Effort”. In: 9th Int. Conf. AOSD f10, Saint Mello, France 2009.

[19] Berg, K. van den, Conejero, J., Hernandez, J. “Analysis of Crosscutting in Early Software Development Phases based on Traceability”. In: Rashid, A., Aksit, M. (eds.). Transactions on AOSD III. LNCS, vol. 4620, pp.73–104, 2007.

[20] Hassan Gomaa. “Designing Concurrent, Distributed, and Real-Time Applications with UML”. Addison-Wesley, Object Technology Series, 2000.

[21] Simon Hurst. “The Characteristic Function of the Student-t Distribution”. Financial Mathematics Research Report No. FMRR006-95, Statistics Research Report No. SRR044-95, 1995.

[22] Boneau, C. Alan. “The effects of violations of assumptions underlying the t test”. Psychological Bulletin 57 (1): 49-64, 1960.

Appendix A

Use Case Description of the Case Study: ATM Sytem

This chapter presents all of the use case description of the ATM system which we created during requirements phase. This system contains six use cases (including use case after change): *Validate PIN* use case, *Withdraw Funds* use case, *Query Account* use case, *Transfer Funds* use case, *Cancel Transaction* use case, and *Borrow Money* use case.

Table A.1: *Validate PIN* Use Case Description

Use-Case Name:	**Validate PIN**	
Use-Case ID:	ATM01	
Use-Case Type:	Concrete Use Case	
Source:	-	
Actor:	ATM Customer	
Description:	This use case describes an event that the system validates customer PIN.	
Precondition:	ATM is idle, displaying a Welcome message.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Customer inserts the ATM Card into the Card Reader	**Step 2:** If the system recognizes the card, it reads the card number.
		Step 3: System prompts customer for PIN number
	Step 4: Customer enters PIN.	**Step 5:** System checks the expiration date and whether the card is lost or stolen.
		Step 6: If card is valid, the system then checks whether the user-entered PIN matches the card PIN maintained by the system.
		Step 7: If PIN numbers match, the system checks what account is accessible with the ATM Card.
		Step 8: System displays customer account and prompts customer for transaction type: Withdrawal, Query, or Transfer.
Alternate Courses:	**Alt-Step 2:** If the system does not recognize the card, the card is ejected. **Alt-Step 5:** If the system determines that the card date has expired, the card is confiscated. **Alt-Step 6:** If system determines that the card has been reported lost or stolen, the card is confiscated. **Alt-Step 7:** If the customer-entered PIN does not match the PIN number for this card, the system re-prompts for the PIN. **Alt-Step 7:** If the customer enters the incorrect PIN three times, the system confiscates the card.	
Postcondition:	Customer PIN has been validated.	

Table A.2: *Withdraw Funds* Use Case Description

Use-Case Name:	**Withdraw Funds**	
Use-Case ID:	ATM02	
Use-Case Type:	Concrete Use Case	
Source:	Including ATM01	
Actor:	ATM Customer	
Description:	This use case describes an event that a customer withdraws specific amount of funds from a bank account.	
Precondition:	ATM is idle, displaying a Welcome message.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 2: Customer selects Withdrawal, enters the amount to be withdrawn.	**Step 1: Include Validate PIN (ATM01).** **Step 3:** System checks whether customer has enough funds in the account and whether the daily limit will not be exceeded. **Step 4:** If all checks are successful, the system authorizes dispensing of cash. **Step 5:** System dispenses the cash amount. **Step 6:** System prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance. **Step 7:** System ejects card. **Step 8:** System displays Welcome message.
Alternate Courses:	**Alt-Step 4:** If the system determines that there are insufficient funds in the customer 's account, it displays an apology and ejects the card. **Alt-Step 4:** If system determines that the maximum allowable daily withdrawal amount has been exceeded, it displays an apology and ejects the card. **Alt-Step:** If the ATM is out of funds, the system displays an apology, ejects the card, and shuts down the ATM.	
Postcondition:	Customer funds have been withdrawn.	

Table A.3: *Query Account* Use Case Description

Use-Case Name:	**Query Account**	
Use-Case ID:	ATM03	
Use-Case Type:	Concrete Use Case	
Source:	Including ATM01	
Actor:	ATM Customer	
Description:	This use case describes an event that a customer receives the balance of a bank account.	
Precondition:	ATM is idle, displaying a Welcome message.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 2: Customer selects Query.	**Step 1: Include Validate PIN (ATM01).** **Step 3:** System reads account balance. **Step 4:** System prints a receipt showing transaction number, transaction type, and account balance. **Step 5:** System ejects card. **Step 6:** System displays Welcome message.
Alternate Courses:	-	
Postcondition:	Customer account has been queried.	

Table A.4: *Transfer Funds* Use Case Description

Use-Case Name:	**Transfer Funds**	
Use-Case ID:	ATM04	
Use-Case Type:	Concrete Use Case	
Source:	Including ATM01	
Actor:	ATM Customer	
Description:	This use case describes an event that a customer transfers funds from a bank account to another.	
Precondition:	ATM is idle, displaying a Welcome message.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 2: Customer selects Transfer and enters amount, and receiver account.	**Step 1: Include Validate PIN (ATM01).** **Step 3:** If the system determines the customer has enough funds in the customer account, it performs the transfer. **Step 4:** System prints receipt showing transaction number, transaction type, amount transferred and account balance. **Step 5:** System ejects card. **Step 6:** System displays Welcome message.
Alternate Courses:	**Alt-Step 3:** If the system determines that the receiver account number is invalid, it displays an error message and ejects the card. **Alt-Step 3:** If the system determines that there are insufficient funds in the customer 's from account, it displays an apology and ejects the card.	
Postcondition:	Customer funds have been transferred.	

Table A.5: *Cancel Transaction* Use Case Description

Use-Case Name:	**Cancel Transaction**	
Use-Case ID:	ATM05	
Use-Case Type:	Concrete Use Case	
Source:	-	
Actor:	ATM Customer	
Description:	This use case describes an event that a customer dismisses current transaction.	
Precondition:	ATM is in use and during a transaction being executed and there is a Cancel button on the current page on the ATM screen.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Customer press Cancel button.	**Step 2:** System dismisses the transaction being executed. **Step 3:** System ejects card. **Step 4:** System displays Welcome message.
Alternate Courses:	-	
Postcondition:	The transaction has been cancelled.	

Table A.6: *Borrow Money* Use Case Description (Addition According to the Change)

Use-Case Name:	**Borrow Money**	
Use-Case ID:	ATM06	
Use-Case Type:	Extension Use Case	
Source:	Extending ATM02	
Actor:	ATM Customer	
Description:	This use case describes an event that a customer cannot withdraw specific amount of funds from a bank account because of insufficient amount of money in the bank account, so the customer requests to borrow the money from the bank.	
Precondition:	There is not enough amount of money in the bank account to be withdrawn according to the customer's required amount.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Customer selects Borrow Money.	**Step 2:** System checks whether the required amount of money does not exceed the maximum borrowing limit. **Step 3:** If check is successful, the system authorizes dispensing of cash. **Step 4:** System dispenses the cash amount. **Step 5:** System prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance. **Step 6:** System ejects card. **Step 7:** System displays Welcome message.
Alternate Courses:	**Alt-Step 3:** If system determines that the maximum allowable borrowing amount has been exceeded, it displays an apology and ejects the card.	
Postcondition:	The remaining customer funds have been withdrawn and the money has been borrowed.	