Addressing Aspect Interactions in an Industrial Setting: Experiences, Problems and Solutions

Autor:
Arturo Zambrano

Director:
Dr. Johan Fabry, DCC, U. de Chile

Co-director:
Dra. Silvia Gordillo, Facultad de Informática, UNLP.

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas
Facultad de Informática - Universidad Nacional de La Plata

Marzo de 2013

Abstract

Aspect oriented programming (AOP) introduces new and powerful modularization constructs. The *aspect* module is used to encapsulate crosscutting concerns, which otherwise would remain tangled and scattered. The idea of encapsulating crosscutting concerns rapidly expanded to earlier phases in the development cycle, including requirement analysis (aspect oriented requirement engineering, AORE) and design (aspect oriented modeling, AOM). The overall application of aspect orientation concepts is known as aspect oriented software development (AOSD).

AOP is not yet a mainstream practice. Particularly AOSD is still in its early stages. This is reflected in the lack of reports of full development cycles using aspect oriented approaches, especially using industrial case studies. Furthermore, the power of aspects comes at the price of new challenges, one of them is that systems built using aspects are more difficult to understand. The crosscutting nature of aspects allows them to alter the behavior of many other modules. As a result, aspects may interact in unintended and unanticipated ways. This problem is known as *aspect interactions*.

In this work we deal with the *aspect interaction problem* in the context of an *industrial domain*: slots machines. We perform a complete development cycle of the slot machine software. This is, to the best of our knowledge, the first complete industrial case of study of aspect orientation. Through this experience we discovered the limitations with regard to aspect interactions, of some emblematic aspect oriented approaches for requirement engineering, design and implementation.

The contribution of this work is threefold. Firstly, we contribute with the evaluation and extensions to some of AORE and AOM approaches, in order to provide explicit support for aspect interactions in requirement analysis and design phases. We also evaluate the implementation of interactions using a static and a dynamic AOP language, and propose an AspectJ extension that copes with aspect interactions. Secondly, this work is the first report of a complete aspect oriented development cycle of an industrial case study. Thirdly, this work provides a complex case study that presents several business logic crosscutting concerns, which in turn exhibit numerous aspect interactions, that serves as a challenging test bed for upcoming AOSD approaches.

Contents

Acknowledgments

First of all, I would like to thank my advisor, professor Johan Fabry, who encouraged me in many ways to finish this work. I have learned a lot by working with him and I honestly expect us to continue publishing together. Secondly, I would like to thank both the professors and students at the DCC - University of Chile. Even though my stays were short I've learned a lot during my time there. I would especially like to thank Renato Cerro for helping with my English.

The travel allowance between La Plata and Santiago de Chile was mainly obtained as part of the Pablo Neruda Program.

I also want to thank my co-advisor, professor Silvia Gordillo and the Lifia laboratory in general for providing me with great freedom to conduct my research.

Part of the experiments reported in Chapter 4 and Chapter 6 were developed in collaboration with Guillermo Jacobson and Alejandro Alvarez respectively. Thanks Guille and Ale!

A big "thanks" to Paul Adamczyk for his useful and detailed comments. I am deeply grateful to all the people who spent time reading this work, asking insightful questions and suggesting improvements: Federico Balaguer, Andy Kellens, Leandro Antonelli and Diego De Sogos. I have done my best to get these comments addressed.

This work could not have been finished without the help and support of all my coworkers at Lifia who covered my back during my trips and, especially this writing.

Last but not least, this could not have been possible without the support of my family: Ana, Lupe, Bruno and the upcoming baby, whose name has not been decided yet :).

Chapter 1

Introduction

Research in software engineering is a continuous quest for more effective and efficient methods for constructing reliable software. Since the early days of software engineering, the criteria and tools for modularization are at the core of research in this field. The seminal work of Parnas [65] established that modules should be considered in function of hiding difficult or changing design decisions. Dijkstra [30] explained the need of dealing with one issue at a time in order to scale the complexity of the system to be built. These ideas guided the evolution of modularity in software engineering. However, the changeable and complex nature of software makes its development, maintenance and evolution still hard.

A complex system is composed of a myriad of concerns. A concern is some part of the problem that we want to treat as a single conceptual unit [86]. Following the modularization concept, software is therefore divided into modules which are supposed to encapsulate the behavior for the different concerns, isolating them as much as possible. Different kinds of modules have been developed *e.g.* functions, procedures, objects. These modules are intended to encapsulate the different concerns that form the system to be developed. The modularization mechanism also includes ways to invoke the behavior defined in the modules (message sends, function invocation, procedure calls) in order to obtain the desired final behavior from a set of collaborating modules.

However, experience has shown that not all concerns can be isolated using conventional modules [33]. These problematic concerns, called *crosscutting concerns* are concerns that do not align with the chosen module structure of the system. Typical examples of crosscutting concerns are derived from non-functional requirements, such as persistence, logging, synchronization, etc. Depending on the different domains, there also exist functional crosscutting concerns, as we will illustrate in this work. Crosscutting concerns affect many different modules of the system. This results in scattered and tangled code, which leads to serious maintainability problems. Because of the former, the code of a crosscutting concern cannot be located easily *i.e.,* it is spread along several units, and therefore it is hard to modify it consistently. Because of the latter, source code units not only contain the code of the intrinsic concern, but also the code of the crosscutting concern. Therefore, evolving the code of the main concern of a module is error prone, as the crosscutting concern code is also there and can be accidentally modified.

In 1997, Kiczales *et al.* [53] presented *aspect oriented programming* (AOP), which is intended as a solution to the problem of encapsulating crosscutting concerns. AOP adds a new kind of module called *aspect*, to encapsulate crosscutting concerns. Pieces of functionality that otherwise would be scattered and tangled with other modules are now contained in one module. Aspects are comprised of *pointcuts* and *advices*. A *pointcut* is a predicate on joinpoints: well defined execution points in a program. An advice defines (part of) the functionality of an aspect, in an similar manner as methods define object behavior. Advices are associated with a pointcut, as a result, when a pointcut matches a joinpoint, the associated advice is executed.

Aspect orientation rapidly gained notable attention of the scientific community, being signaled by the MIT in 2001 as a one of the key technologies for the next ten years [89]. Somewhat surprisingly, it has however not been widely accepted by developers, and arguably did not become a widely used technology. This is reflected in a low number of projects, particularly in the industry, using aspect orientation. In this regard aspect orientation can be considered as still being in its early stages and there are several problems that remain as impediments for its adoption. In this work we treat what we consider an important one: the aspect interactions problem. Aspect interactions are composition issues that occur when the behavior or structure of two or more aspects interfere, either in a possitive or negative way.

1.1 Unfinished Business I: Aspect Orientation in the Industry

Even though aspect orientation offers new features for advanced modularity, which still is a problem in software development, it has not been massively adopted. That is, most of the software development being done does not use aspect orientation. Muñoz *et al.* in [61] report that less than 0.5% of the projects written in Java that were added to SourceForge between 2001 and 2008 use aspects.

Besides the deficiency in the number or proportion of applications, aspect orientation is usually associated with the implementation of non-functional crosscutting concerns, such as validation, authorization, security, persistence and logging, as reported in [68], which is a survey of industrial projects covering 8 developments in different 3 domains. Furthermore, in that report it is argued that AOP is used initially to address development concerns such as the enforcement of architectural restrictions. Only later the relatively basic concerns mentioned above are considered. In this way, the scope of applications for aspect orientation can be considered somewhat restricted. Notably, core (business logic) concerns are not considered for their implementation as aspects. Since aspect orientation is still evolving, the application of its concepts to a broader set of domains and especially to core functional crosscutting concerns is desirable, in order to obtain more general and practical solutions.

From the projects listed in [68] only the Toll system [73] has been developed using aspect orientation concepts starting from the requirement analysis phase. However, we found no complete and detailed report of this experience, as the authors focused mainly in requirements analysis for this work. To the best of our knowledge there is no report of a complete development cycle of an industrial complex system using aspects. We consider this absence a significant deficit in aspect orientation research. This work is a step in that direction, presenting a case where all the concerns, their behavior and interactions are derived from industrial requirement sources.

1.2 Unfinished Business II: Aspect Interactions

Aspect orientation presents an inherent trade-off between simplicity and expressiveness. The powerful features of aspect orientation come at the price of complexity in understanding the behavior of the whole system. That is, the crosscutting nature of aspects makes it difficult to reason about their impact. Besides this, due to the fact that aspects potentially affect several elements in the base program, chances are that they interfere in some way.

Aspects introduce their behavior at certain execution points (joinpoints). Aspects may interact by applying their behavior at the same joinpoint, resulting in such interference at joinpoint level. As we show in Sect. 2.2, most of the research we have encountered is fundamentally devoted to the detection and, to a lesser extent, the treatment of joinpoint interactions. In addition to joinpoint interactions however, semantic interactions can also occur. In these interactions, an aspect may depend on the actions performed by other aspects, that are not necessarily present at the same joinpoints. For example, an aspect can collect information regarding user preferences. Other aspects can use such information to improve or personalize the user experience. Alternatively, this may be because an aspect uses structure or behavior installed by another aspect. There could be also incompatible behavior interference of aspects, leading to semantic conflicts, for example due to concerns that present a trade-off.

In some cases, interactions are desirable, and their occurrence must be ensured. For example, the case of *dependencies*, where an aspect requires another aspect. Consider a distribution aspect that needs data to be encrypted by another aspect. In other cases, where aspects provide incompatible behavior, interactions must be avoided. For example a caching aspect that directly returns a method's result can bypass an authorization aspect, resulting in a system where users access information they should not get [39]. In this case it is said that there is a conflict between these aspects.

To the best of our knowledge, the aspect interaction problem is still largely unsolved, and remains a matter of investigation, as mentioned in recent research work [58, 68]. Besides focusing on joinpoint interactions and specifically on the detection of unexpected interactions, existing research work on aspect interactions however treats them in only one phase of the development cycle.

We have found development approaches that support interactions at requirements analysis, design and implementation but none that cover interactions along the whole development cycle.

1.3 The Slot Machine Domain

For this work we have chosen the slot machine (SM) domain. A SM is a gambling device typically found in casinos. It usually has five reels which spin when the *play* button is pressed. The SM has pre-configured prizes, which are paid according to the symbols shown at the end of the spin. We have 5 years of experience in this domain, working full-time in programming both the management applications (accounting) and the SM games. We therefore can claim to have acquired deep knowledge of this domain.

Beyond our knowledge of it, the case study chosen for this work has interesting characteristics regarding the issues mentioned above:

- It is derived from an industrial domain, with publicly available requirements documentation.
- The sources of these documents are many organizations or institutions with different (legal, technical, business) backgrounds. As they are independent sources, they release the documentation at different speeds, leading to a nontrivial example of evolution of requirements.
- Our experience with this domain has taught us that there is a significant amount of crosscutting concerns in these applications. Furthermore, these concerns depend on, and interact with each other as well as with the modularized concerns.
- Following the taxonomy of aspect interactions of Sanen *et al.* [74], we have found that it has several examples of each interaction type (as will be shown in Sect. 3.8).
- The interactions have considerable impact on the game behavior. In our experience, not having proper mechanisms for handling them has been the source of costly bugs.

1.4 Motivation

The full potential of aspect orientation will not be unleashed until the problems of applicability to real world cases and aspect interactions mentioned before, are addressed. Case studies that explore the capabilities and limitations of existing approaches need to be carried out. Aspect interactions must be studied in the context of real world problems. This understanding is a prerequisite to develop aspect interaction traceability support.

We therefore decided to perform a complete development cycle of a nontrivial system: the slot machine software. This serves as a vehicle that allows us to study aspect interactions modeling and implementation during development, and to contribute to their explicit support.

1.5 Thesis Statement

Complex systems, such as slot machines, are composed of several functional and non-functional concerns. These systems, when developed using aspect orientation, can present semantic interactions between the aspects. These interactions need to be controlled in order to provide the desired behavior. Hence a complete aspect oriented development cycle of such systems requires aspect interaction support. Unfortunately, this support is not currently available. We claim that tracking aspect interactions from the requirements to the implementation phase should improve our understanding of their nature, allowing us to develop new tools and more effective strategies for coping with them.

1.6 Objectives

To address the need for case studies of complete development cycles of nontrivial aspect software with dependencies and interactions, this dissertation undertakes the development of an industrial case. We use aspect orientation with a focus on aspect interactions from the onset, in order to improve the understanding of interactions, identify where current aspect oriented approaches are lacking, and contribute extensions to enable interaction support.

We can summarize the derived objectives as follows:

- Study an industrial case featuring several interacting crosscutting concerns.
- Understand the interactions between these concerns.
- Assess the expressive capabilities of existing aspect oriented requirement analysis tools for interactions and extend them as necessary.
- Judge the support provided by aspect oriented modeling approaches.
- Evaluate how the designed interactions can be implemented in different aspect oriented programming languages.

1.7 Methodology

The SM software exhibits numerous interacting crosscutting concerns and the requirements change at different speeds. If a requirement corresponds to a single concern that was neatly encapsulated in a single module, the effect may be controlled and more easily followable using conventional software engineering approaches. On the other hand, if a requirement belongs to a crosscutting concern that interferes with other crosscutting concerns, dedicated interaction support is needed to trace it across several stages in the development cycle. We therefore have opted to use Aspect-Oriented Software Development, taking special care of dependencies and interactions between the different aspects and modules. Being aware of the critical importance of interactions in this domain, we have focused on interaction modeling and implementation. In the requirement analysis and design phases several approaches have been tried out in order to express the interactions as part of our models. During the implementation phase two programming languages have been tested to implement these interactions. We now discuss this in more detail.

We focused on interactions early in the development cycle, at the requirement analysis phase. This was motivated by the argument of Liu *et al.* [57] that most feature interactions can be detected in early stages of the software development cycle by reasoning on the causes of interactions and building models of them. The objective of this detection is to document as many interactions as possible so that this information can be used later in the design and implementation phases.

The result of the modeling process should therefore be a model of the requirements that is as consistent as possible. As complete consistency is not possible in the presence of certain conflicts, we want to document them to defer their resolution to later development phases in these cases. Hence, to accomplish this objective, it is necessary to be able to rely on expressive mechanisms in the selected modeling technique for this phase.

To the best of our knowledge there is no previous work detailing experiences with AORE approaches and their interaction support in a large-scale industrial case. We therefore opted to perform an in-depth study of two approaches to evaluate their applicability in the slot machine domain: Theme/Doc [9] and MDSOCRE [60]. The results revealed deficiencies that were tackled by extending them. One of the extensions was tested with engineers showing that it was both more accurate for representing interactions and more efficient in terms of time.

The next step was modeling the software using an adequate approach for *Aspect Oriented Modeling* (AOM). However, to the best of our knowledge there has been no work published that evaluates AOM approaches in an industrial setting with a focus on interactions between the different concerns. We therefore undertook an evaluation of two mature AOM approaches to establish their applicability in our context: Theme/UML [26] and WEAVR [28].

In the design phase, our goal was to refine the requirement specification documents into a model of the software artifacts that will form the final system. This model, written down in a design document, would then be passed to the developers for implementation. Hence, it should be sufficiently complete to allow for the implementation to be produced relatively independently. We expected to be able to produce the complete design documents, *i.e.,* not having to resort to a significant amount of additional documents with an ad-hoc notation to complement for omissions in the methodology. In the latter case, the advantages of using a standard AOM are small and we would consider rolling our own. We furthermore have two related expectations of the design document: maintenance support and explicit interactions.

In subsequent maintenance or evolution phases, the changes made in the requirements will trigger subsequent changes in the design, and the developers will modify the implementation accordingly. Such later modifications must not break the system because they violate constraints of the original design or go against the original design decisions. It is known that the presence of aspects in a software system that is evolving can be problematic [51]. Such issues should be mitigated by the information that is explicitly available in the design document.

Aspect interactions may hinder the understanding of the expected behavior, information that is crucial for the correct implementation and evolution of the system. Documented design decisions should therefore include not only which modules will be aspects and where they crosscut, but also how they *interact* with each other. This information must be made explicit so that critical information is correctly passed to the implementation phase, and is present when maintaining or evolving the software.

Once the design decisions regarding the interactions have been somehow established during the design phase, an implementation must be performed following the design directives. In this stage it is desirable to rely on language constructs that allow the programmer to explicitly code the interactions. We therefore reviewed the existing support for interactions and decided to undertake the implementation of interactions using general purpose aspect languages, in order to evaluate the impact they have on such implementation. First we chose AspectJ [52], which is arguably the most mature and influential aspect language. As AspectJ is a static language, we decided to contrast the results with the implementation of the interactions using a dynamic aspect language, in this case we chose PHANtom [34].

Implementing interactions in such languages is an important task in order to assess the convenience offered by their features when developing ad-hoc logic for interaction resolution. This stage also provided some interesting insights, such as how some interactions benefit from the static (compiler) checks of AspectJ, while others take advantage of the dynamic weaving of PHANtom. Furthermore, this experiment helps to determine the weak points of these languages and possible extensions.

1.8 Contributions

We can summarize the contributions of this work grouping them as follows:

- Aspect Interactions:
    - Evaluation of two AORE approaches.
    - The extension of these two approaches providing explicit support for aspect interactions.
    - Evaluation of two AOM approaches.
    - The implementation of the four types of interactions (conflict, mutex, dependency and reinforcement [74]) using both a dynamic and a static aspect oriented language.
    - Based on our experience, the proposal of extensions to the AspectJ language in order to furnish it with explicit interaction support, covering our interactions and also other typical interactions reported in the literature.
- Industrial Case Study:
    - The first report of the development cycle using aspect orientation in requirements analysis, design and implementation in the context of an industrial test case.
    - The identification and documentation of examples for the four types of interactions (conflict, mutex, dependency and reinforcement) in the context of an industrial test case, that serves as a challenging test bed for upcoming AOSD approaches.

1.9 Outline of this Dissertation

Following the ideas presented previously, this dissertation is organized in the following way:

- Chapter 2 presents the necessary background on aspect orientation and the related work on aspect interactions for the three treated phases of the development cycle (requirements analysis, design, and implementation).

- Chapter 3 describes our domain and some peculiarities that makes it an worthwhile case study. At the end of this chapter we describe the concern decomposition used along this work, and the interactions found between concerns.

- Chapter 4 presents the evaluation and results of applying two aspect oriented requirement analysis approaches: Theme/Doc and MDSOCRE. This chapter also presents extensions for the approaches and the evaluation of the modified approaches compared with original ones.

- Chapter 5 shows the study of two recognized aspect oriented modeling approaches and how the interactions have been expressed using their capabilities: Theme/UML and WEAVR.

- Chapter 6 presents the results of the implementation of the four interaction types for both a static (AspectJ) and a dynamic (PHANtom) aspect language. The effect of the nature of these languages is discussed and compared, and an extension for AspectJ is presented.

- Finally, Chapter 7 presents our conclusions from the overall experience and several lines of future work for the treatment of aspect interactions during the different stages in the development cycle.

Chapter 2

Aspect Oriented Software Development and Aspect Interactions

In this chapter, we provide the necessary context and background on research that motivates this thesis. We briefly discuss the modularity problem that gives rise to advanced modularization approaches, particularly aspect orientation. We also present the *aspectual interactions* problem and review the research on aspect interactions throughout the software development cycle.

2.1 From Objects to Aspects

2.1.1 Modularization Issues in Current Software Engineering Practices

Concerns are sets of information that have some impact on the system. Typical examples of concerns are business logic, synchronization, real-time constraints, error detection and correction, data validation and persistence. Software systems are therefore composed of a myriad of concerns. All these concerns need to be considered during the development process. Early on, it was observed that different concerns need to be treated one at a time.

As Dijkstra stated in [30]:

> " ... one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects."

Software engeering follows this idea, breaking a big problem into smaller problems, solving them and assembling the results. In order to reduce the complexity of software being developed, different modularization paradigms are applied to obtain loosely coupled software artifacts, each addressing a separate concern. These paradigms provide mechanisms to assemble resulting modules functionality. For example, object oriented programming breaks solutions down into object or classes, while functional programming breaks them down into functions. Functionality is then assembled by *message sends* in object orientation, and *function invocation* respectively.

In spite of its prevalence, object orientation is not the silver bullet of software engineering. There are concerns that cannot be cleanly encapsulated into one class. On the contrary, their functionality cuts across many classes in the system, making their implementation difficult to track and maintain. This problem is known as scattered code. Furthermore, the code of these concerns obscures the code of the classes where they are applied. This problem is known as tangled code. The concerns that not cleanly fit into the underlying paradigm constructs and cut across other modules are called *crosscutting concerns.* Typical examples of this kind of concern are: caching, logging, monitoring, etc. Usually crosscutting concerns are related to non-functional concerns, but this is not a rule. As we will see in Sect. 3.7, in our case study there are also many examples of functional (business logic) crosscutting concerns.

2.1.2 Advanced Separation of Concerns and Aspect Oriented Programming

During the ninties, the modularization problem posed by crosscutting concerns causes the birth of several approaches, which are collectively known as *avanced separation of concerns* techniques. They include: Composition Filters [12], Multidemenstional Separation of Concerns [87], and Aspect Oriented Programming [53] among others.

The term "aspect-oriented" was coined by the team of Gregor Kiczales at Palo Alto Research Center, where the first and most influential aspect language: AspectJ [52] as been developed.

Aspect Oriented Programming (AOP) introduces a new module called an *aspect*, which defines behavior (advice) that is added to the system at specific execution points, called joinpoints. Examples of joinpoints are a method call, a variable assginment. To allow referencing multiple joinpoints at the same time, aspects also define predicates on joinpoints, called pointcuts. Aspectual behavior is executed when a pointcut captures a part of the behavior of the program. According to Filman *et al.* [37]:

> "AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers."

Aspects allow the modularization of crosscutting concerns since they are able to define both the crosscutting behavior (advice) **and** where it needs to be applied (pointcuts).

In AOP, a program can be seen as a flow of joinpoints that represents steps in the execution of the program. Pointcuts can refer to several joinpoints. and advices are associated with pointcuts. In this way, whenever a pointcut matches a joinpoint, the associated advice is executed. Aspects (in most AOP languages) are also able to augment the structure of the base, for example declaring new instance variables in a class. This feature is known as *intertype declarations* or *introductions.*

The code where aspects are applied is called the *base program.* Aspect code is *woven* into the code of the base program by the *aspect weaver.* At a more abstract level, it is said that aspects are composed with the base program. Note that aspects are usually are also able to cut across other aspects.

2.1.3 Aspect Oriented Software Development

AOP concepts gained acceptance during the early 2000's and were extended to other phases of the software development cycle. The set of these techniques are collectively known as aspect oriented software development (AOSD).

In the early stages of software development, requirement analysis is done through Aspect Oriented Requirement Engineering (AORE), which is also known as Early Aspects. AORE is aimed at identification, specification and representation of crosscutting concerns at requirements level.

Aspect Oriented Modeling (AOM) is the part of AOSD concerned with the design of software. Its objective is to specify the structure and behavior of aspect oriented software at an abstract level. These models should be independent from language specific features.

AOSD has arguably not mature enough to become a mainstream paradigm for software development. We believe that one of the causes for this is the inherent difficulty in understanding the behavior of the composed system. This is due to the fact that when analyzing the behavior of a given module it must be considered that aspects may alter it. Furthermore, may aspects may introduce these alterations, causing potentiality harmful (unintended) interferences.

2.2 Interactions in Aspect Oriented Software Development

Aspect interactions are a well know problem [3, 31, 59, 74, 82]. Since the inception of AOP, it has been recognized that the application of multiple aspects at the same joinpoint can yield to undesired effects during program execution. This is also known as shared joinpoint interaction [58]. This differs from semantic interactions where it is not necessary to have several aspects applied to the same joinpoint in order to observe interferences [13, 62].

One typical example of an interaction is related to weaving ordering. Consider the example in Listing 2.1 taken from [88], where the Counter aspect counts the accesses to attributes. This aspect captures execution of setters (line 3) and increments a counter it introduces as new variable in `Point` class (line 2). The Counter aspect can be applied to the class `Point` (lines 8 to 13). Now consider the aspects ThreeD (lines 15 to 18) and Color (lines 20 to 23). These two aspects also introduce new variables in the the Point class and the corresponding setter. The order in which this three aspects are applied (weaved) into the code of the `Point` class is indeterminate. As a result, different weaving orders lead to programs with different behavior, since the new members introduced by the aspects ThreeD and Color may or may not be counted.

Listing 2.1: Aspect interaction example: weaving order.

```
1 aspect Counter {
2   int Point.counter = 0;
3   after(Point p) : (execution(* Point.set*(..))  && target(p)
4     { p.counter++; }
}

class Point {
 int x;
  void setX(int v) { x = v; }
 int y;
  void setY(int v) { y = v; }
}

aspect ThreeD {
 int Point.z;
 void Point.setZ(int v) { z = v; }
}

aspect Color {
 int Point.color;
 void Point.setColor(int c) { color = c; }
}
```

The programmer thus needs tools for controlling the weaving order and consequently select the desired behavior. As a result, it comes at no surprise that most AOP languages and frameworks allow for some kind of control in the order of application of aspects.

As an illustration of shared joinpoint interaction, consider the following two aspects: encryption and validation. Lets suppose that these two aspects are applied on the call to the `setPassword(String)` method. The encryption aspect takes the parameter and encrypt it. The validation aspect checks that the parameter is equal to the expected password. If the encryption aspect is applied before the validation the system will misbehave, as the comparison will not be performed between the actual value of the parameter and the expected password. Instead, it will be done between the encrypted version of the parameter and the expect string.

Besides joinpoint level interactions (more than one aspect applying on the same joinpoint), there are more abstract or high level ones. These interactions are known as semantic interactions [58], and they arise when the presence of an aspect invalidates some assumption of other aspects for example.

The detection, understanding and resolution of interactions is a complex task. Since aspects apply across the whole system (including other aspects), there can be many of them, they can alter the system's state and chances are that an aspect influences other aspects. That is, aspects can interact by affecting directly one another, or indirectly by affecting the (shared) base program.

Bakre *et al.* classify interactions among aspects and base program into three cases: spectative, regulative and invasive [7]. Spectative aspects observe the state of the system at joinpoints, but they do not control the execution flow. Regulative aspects observe the state at joinpoints and control the flow at joinpoints based on the state. Invasive aspects, in addition to being regulative modify the system state at joinpoints. This classification may serve to understand how aspects behaves regarding the base but it is not rich enough to model aspect to aspect interactions.

Marot enumerates four categories for interactions: (1) shared joinpoint interactions, (2) scope interactions, (3) dependency interactions and (4) implementation semantic interactions [58].

Shared joinpoint interactions are subdivided into data-flow and control-flow interactions. In the former an aspect modifies data that is accessed *in the same joinpoint* by others aspects. In the latter the execution of an aspect either bypasses the execution of other aspects by not calling `proceed` or adds more applications by repeatedly calling `proceed`.

Scope interactions include the cases when (1) aspects produces new joinpoints, (2) aspects remove potential joinpoints and (3) aspects having mutually exclusive scopes.

Dependency interactions include (1) structural dependencies: when an aspect uses structural elements introduced by other aspect, (2) control flow dependencies: when an aspect needs to be applied in the control flow of another aspect in order to work correctly and (3) inter-dependent introductions: when an aspect tests the existence of elements introduced by other aspects, in order to add its own structural elements [44].

An implementation semantic interaction occurs when an aspect refers to part of the base system (or another aspect) whose semantics is affected by other aspects, which in turn invalidate some implicit assumption. This may result in harmful interferences.

Sanen *et al.* introduce another taxonomy [74]. In this work, the following types of interactions are defined:

Conflict This denotes semantic interference between aspects. If there is a conflict between aspect A and aspect B, each one can work correctly on the base system, but both cannot not be deployed at the same time. The combination of aspect A and aspect B results in undesirable behavior of the system.

Mutex In this case, two aspects provide similar functionality but they can not be deployed or work on the system at the same time. It is similar to the previously mentioned *conflict*, in the sense that if there is a *mutex* between A and B, both cannot work at the same time.

Dependency There is a dependency of aspect A on aspect B if aspect A explicitly needs aspect B to be deployed. If aspect B is not present, aspect A cannot behave correctly.

Reinforcement There is a reinforcement of aspect B on aspect A if the presence of aspect B benefits the behavior of aspect A. Aspect A can provide the expected behavior even in the absence of aspect B but, if aspect B is present aspect A is able to provide extra functionality.

This taxonomy is the one that better describes the relationships between concerns in our domain, as we will show in Sect. 3.8.

In the following sections, we review the related work on aspect interactions for three software development phases: requirements analysis, design and implementation.

2.2.1 Interactions in the Requirements Analysis Phase

Aspect-Oriented Requirements Engineering (AORE) addresses the requirements engineering problem that some requirements are difficult, if not impossible, to isolate into separate modules. AORE performs first-class modeling of these crosscutting concerns as aspects, identifying and characterizing their influence on other requirements in the system [60, 69]. These models enable better identification and management of requirements conflicts, irrespective of the crosscutting nature of the requirement. Ideally, the result of this phase is to have a consistent as possible model of the system early in the software development life-cycle. As a result, the system may be designed and built correctly when considering the needs of the various stakeholders.

Surveys We have encountered two comparative studies for AORE approaches but these however do not consider aspectual interactions. Sampaio et al. [72] analyze the speed of the requirements analysis process and the quality of the output. Their real world example: 19 pages of requirements specification, is considered almost 10 times smaller than the requirement documents we face, and whose summary is presented in Sect. 4.2. Chitchyan et. al. [21] use several comparison criteria: identification of concerns, composability, decision support, traceability, evolvability and scalability. This work is more conceptual, as it compares the approaches without applying them to a concrete example and instead it considers the mechanisms provided by each approach in light of the criteria mentioned before.

AORE approaches that consider *conflict resolution* as part of their methodology [16, 70] help stakeholders decide on which concern to implement. They however do not consider cases where *all* conflicting concerns must be implemented. In these cases, conflicts need to be documented so that interactions are considered at design time.

In [93] Whittle et al. present an approach called MATA, based on model transformation where weaving is viewed as a special case of graph transformation. MATA provides support for conflict and dependency detection, based on critical pair analysis. The objective of this detection phase is to order composition. More recent work by Chitchyan et al. [19] moves the focus towards a semantic analysis of requirements. Here, requirements are annotated and then composition rules can be expressed using semantic queries. This approach enables automatic detection of certain conflicts, with the aim of removing them.

For the analysis phase (explained in Chapter 4), our focus will lie in interactions between aspects, and capturing them at requirement level. Multiple approaches for capturing requirements using aspects exist that do not provide support for interactions [43, 48, 92] we therefore do not include them here. Some other approaches, such as AORA [15] provide documentation of dependencies, but nothing is said regarding mutex or reinforcement interactions.

The EA-Miner tool from Sampaio *et al.* [73] uses natural language processing techniques for partial automation of the identification of elements in the requirements, which leads to early aspect identification. EA-Miner is worth mentioning as it deals with the ambiguity problem which is a relevant problem in our case. This tool is intended to be of use for any AORE approach and it produces an output that can be consumed by other tools, for example ARCADE [70], which assists in the detection and resolution of conflicts. EA-Miner uses semantic tags to avoid the ambiguity problem. Nonetheless, it requires some interaction with the requirements engineer, especially in the presence of (noun-)phrases.

MDSOCRE is an AORE approach based on XML syntax. It allows for the organization of requirements into concerns. Concerns can be connected by means of *compositions rules.* Composition rules express crosscutting relationships among concerns, with requirement level granularity. These rules can be parameterized in order to specify the semantic of the relationship.

Theme/Doc [9] is a part of a more encompassing approach called Theme [26]. Departing from a set of requirements, Theme/Doc provides heuristics for defining concerns and their responsibilities. Requirements are then attached to concerns. Requirements attached to more than one concern may indicate the presence of a crosscutting concern. Theme/Doc prescribes how to find crosscutting concerns in this situation. It includes a graphical notation at different levels (called "views"), for denoting requirements, concerns and crosscutting concerns. It also provides steps for evolving from requirement analysis to the design phase, called Theme/UML, which we briefly describe in the following section.

Goal oriented requirement engineering *e.g.* [64, 90, 96] deals with conflicts between goals (especially soft goals). A goal captures, at different levels of abstractions, the objective of the system [91], for example: *provide ubiquitous access to the system.* Goals are usually optative. Yu *et al.* [96], propose a systematic process to discover aspects from a goal model containing functional and non functional goals. In this case, conflicts arise during the iterative process of refining a goal graph, and they are solved by removing relationships that lead to a conflict. Related to this is the research on non-functional requirements (NFR). NFRs cannot be fully satisfied, instead they can be "sufficiently" satisfied [23]. That is, non-functional goals can only be satisfied within acceptable limits [22]. The NFR framework establishes different types of contributions between goals. Goals and contributions are associated with labels indicating the degree of satisfaction or denial. These labels are propagated to determine the effect of different design decisions [24]. The engineer can then use this information to make design decisions reflecting the best trade-off for a NFR.

2.2.2 Interactions in the Design and Modeling Phase

The objective of Aspect Oriented Modeling (AOM) is to provide developers with general means to express aspects and their crosscutting relationships onto other software artifacts at design level.

AOM approaches support the specification of base and crosscutting concerns, intertype declarations and other distinctive concepts of aspect orientation. One of the research topics of the AOM community is *model based aspect interference and composition management*, as stated in the *AOM Workshop Series* site [27].

There are several AOM approaches, but few of them claim to support conflicts and other types of interactions. When supported, interactions are generally treated at joinpoint level. That is, two or more aspects working on the same joinpoint.

Wimmer *et al.* authored a survey of AOM approaches [94] where concern interactions are part of the evaluation framework. It shows that most of the surveyed approaches do not provide for interaction support. Of those that do, most focus on detection of syntactic and semantic interactions. A representative approach is to transform UML models into graphs which are then analyzed to look for interactions. This approach is also advocated by Ciraci *et al.* [25] and Mehner *et al.* [59].

Another example of detection of semantic aspect interactions is presented by Mussbacher *et al.* in [62]. This approach works on top of any kind of model with a well defined meta-model. In this approach the model elements are tagged with domain specific labels, called semantic markers. These markers are used to calculate the effect of aspects on the base and between them. The markers are backed by a goal model which can evaluate the contribution between the different markers. The model is composed using the underlying modeling language. After composition, the elements of the goal model are instantiated, and goal model analyzed looking for potential conflicts.

Similarly, detection of interactions in the design phase has been considered in the feature oriented programming community, *e.g.* the work of Apel *et al.* on FeatureAlloy [3] detects structural (syntactic) and semantic dependencies as well.

The basic assumption in all the above is that interactions are unintended and arise during aspect composition. This however does not hold in our case as interactions may be planned and moreover, already will have been detected during the requirements phase. Instead of detection, we need the design to effectively document the decisions made to manage them.

Other authors purely focus on avoiding interactions. For example, Katz and Katz describe how to build an interference-free aspect library [50]. In our case however, some interactions are required to obtain the desired behavior, and other interactions cannot be removed, but should be controlled instead.

Theme/UML [26] is the second part of the Theme approach. It provides support for separately designing the different concerns (called *themes*) identified during the analysis phase. Each theme defines its own class diagram and its behavior is represented using sequence diagrams. For providing crosscutting support, sequence diagrams are parameterized in order to define where crosscutting behavior needs to be attached. This parameters can be bound to the actual objects and method calls, so that crosscutting behavior is added. Theme/UML claims to handle conflicts between themes.

WEAVR [28, 29] is a modeling language for MDE, which supports aspect orientation. WEAVR is an extension for other MDE tools used by Motorola to develop telecommunications software. It uses state machines written in the Specification and Description Language (SDL) [47], for expressing the behavior of the system. It supports aspect orientation, as it allows weaving together different state machines. Crosscutting state machines define action and transaction pointcuts, which are used to express where the aspectual behavior must be added. Furthermore, WEAVR supports conflict resolution and dependency management.

It is important to note that the vast majority of AOM work on interactions refer to dependencies and conflicts, but neglect or minimize reinforcement or mutex. This may indicate that these types of interactions are considered less frequent. However, they nonetheless occur in the context of our work, and we see no reason why it would be an exceptional case.

2.2.3 Interactions in the Implementation Phase

Ideally, aspects are programmed independently, and later composed with the base. As aspects implement different views of the system, it should be possible to select which aspect must be woven for a given build of a system, enabling only the desired functionality. However, once they are composed, aspects interact in different ways, making it difficult to understand of how the final system will actually work.

Aspect interactions have been treated in different aspect oriented programming languages and frameworks [88]. Unfortunately, this has been done mostly considering interactions only at joint point level.

Two tendencies can be identified in research regarding the treatment of aspect interactions and programming languages. Firstly, most of the research work on interactions has been devoted to the detection of them. The purpose of this work is to raise programmer awareness regarding unexpected interactions, so that he can introduce changes in the system to obtain the desired behavior. That is, he can consciously confirm or avoid them. Secondly, some work deals with the resolution of the interactions. That is, how to control multiple advice execution when more than one aspect applies on a single joinpoint. These control mechanisms include precedence, reordering of advice execution, conditional execution of certain advices, etc.

In the context of this work we have particular interest in those approaches or tools that provide some kind of built-in support for the resolution of aspect interactions.

Douence *et al.* proposes *A Framework for the Detection and Resolution of Aspect Interactions* [32] which is based on the notion of separating the aspect interaction treatment from the aspect definition. The model proposes three phases:

1. Programming: aspects are written independently.
2. Conflict Analysis: automatic detection of interactions among aspects.
3. Conflict Resolution: the programmer must resolve the interaction using dedicated language constructs for detailed composition. The result can be analyzed for conflicts detection once again going back to 2.

The approach is based on a runtime infrastructure that observes the execution and inserts behavior as needed, based on the execution state of the base application.

A formal aspect language is defined that allows for crosscuts definitions, and a very specific semantic for weaving aspectual behavior. In this approach two aspects are considered independent if they do not share a joinpoint (their crosscuts do not match the same joinpoint). If the aspects are independent, it is sufficient to say that the resulting program is not affected by the order of aspect weaving. If the aspects are not independent, the programmer is required to resolve the interactions. Two kinds of "independence" are considered in this approach:

Strong independence which means that aspect independence is held for every program. On the positive side this property does not need to be re-checked after a program modification. On the negative side it is a strong condition.

Independence w.r.t. a program this independence takes into account the possible execution traces of a program. This property is a weaker condition to enforce, compared against *strong independence*.

Once a conflict (a shared joinpoint) has been detected it needs to be resolved by the programmer. The language proposed by Douence *et al.* allows for the composition specification of each insert, in each conflicting joinpoint, but this is admitted to be a tedious task. For convenience, the language offers constructs allowing to specify composition at aspect level that can be used automatically whenever a conflict arises. For example, if aspects A1 and A2 conflict, it is possible to express that only aspect A1 inserts will be applied, or specify the order of weaving of the insert (using the *seq* function) to indicate that A1 inserts come before those of A2.

Re-ordering of advice weaving for a given joinpoint is the principal mechanism offered by this approach in order to cope with interactions. The formalism and overhead of work for this approach can be however considered an impediment in an industrial context.

Reflex [84] started as a library providing partial behavioral reflection, and evolved into an AOP kernel capable of supporting different aspect oriented languages. Its architecture features three main layers:

- Transformation layer: based on a reflective core extending Java with new structural and behavioral reflective capabilities.
- Composition layer: responsible for the automatic detection of aspect interactions. This layer provides explicit means for resolution of such interactions.
- Language layer: allowing for the embedding of different (domain specific) aspect languages.

The central notions behind the Reflex architecture [35] are:

Links Behavioral transformation is done through the use of *Links*, which are responsible for binding a set of program points (*joinpoints* in AOP) to a meta-object. There are *structural* and *behavioral* links. A behavioral link is characterized by the meta-object in control, the *activation condition* and other attributes. On the other hand, structural links bind structural or behavioral actions to structural cuts (joinpoints), therefore they can be used, for example, to implement the *inter-type declaration* of AspectJ. As links bind joinpoints with a single action in a meta-object, they can be seen as a primitive aspect (containing just one advice).

Hooksets A hookset corresponds to a set of program points, or static cuts. They are specified as predicates matching reified program elements. Hooks are inserted, according to a *hookset definition*, as part of the behavioral links installation process, which occurs at load time.

Meta-objects Meta-objects implement the aspectual behavior, that is, they are the equivalent to an *advice* in AOP terms. Any Java class can be used as meta-object in Reflex, as their protocol can be adapted and called from the corresponding link.

A link binds a hookset to a meta-object. According to the specification of the link, program information is reified. Behavioral links are setup at load time, while structural links are applied at load time. Note that structural links application is performed before behavioral links setup, as structural changes can be subject to behavioral cuts (an *structural dependency* in the context of Marot's taxonomy [58]) .

Reflex supports the *detection, resolution and composition* proposed by Dounce *et al.* in [32]. The AOP kernel detects the interactions, which are then notified to the *interaction listener*. The default interaction listener warns the programmer of unintended interactions.

Other listeners can be defined, so that they handle the interaction appropriately. Interactions may occur for behavioral and structural links. For behavioral links, there is an interaction if a joinpoint is captured for more than one hook set. For structural links, there is an interaction if a class is loaded by more than one class set.

Reflex supports mutual exclusion between links, thanks to the *link interaction selectors*. When a link is involved in an interaction, the *selector* is queried in order to determine if the link actually applies or not (the result of this calculation also depends on the other links present in the interaction).

The resolution process consist then into two steps. Firstly, links that should be applied are selected. Secondly, links are ordered or nested.

The mutex support of Reflex seems to be appropriate for mutex interactions at joinpoint level. The kernel low level operators might be used to implement new constructs to provide certain support for the conflicts in SM. Reinforcement and dependency, as they occur in our case study domain, are more high level interactions and therefore cannot be supported by Reflex. Unfortunately, Reflex is not being actively maintained anymore.

Phase [58] is an aspect oriented programming language built on top of Pharo Smalltalk, whose main characteristic is its reflective architecture and meta joinpoint model. Phase supports aspects reflecting upon reflective aspects, which is a fundamental characteristic of reflection. Phase has been built to prove the hypothesis that empowering aspects so that they can observe, augment and modify (part of) the execution of other aspects is relevant. In order to do this, Phase provides a powerful and expressive meta joinpoint model, which includes joinpoints referring to aspect weaving and execution.

This meta joinpoint model allows for the detection and resolution of shared joinpoint, scope interaction, dependencies and interdependent introductions. For example: the programmer can set specific precedence in joinpoints where other precedence policies are already defined, in order to solve conflicts.

The detection of interactions in this reflective AOP environment can be implemented as aspects that observe the aspect-level execution in order to report aspect composition issues. That is, there are dedicated aspects which are looking for events that invalidate the desired properties of the aspect system. As these events are specific to each interaction case, "detection" aspects are also specific.

Once the problem is detected, a resolution is provided for each specific case. Detection is possible thanks to the already mentioned meta joinpoint model.

Phase is a proof of concept of reflective AOP languages and has been only applied to small size examples, which is not our case.

PHANtom Other Smalltalk based aspect languages include AspectS [46], which is not being actively developed, and PHANtom [34]. PHANtom is a new general purpose aspect oriented language supporting the core AOP constructs (aspect, advice, dynamic and static pointcuts, inter-type declarations, etc.), plus extended features such as computational membranes used to order of execution of aspects and provide reentrancy control. There is no special syntax for PHANtom, instead programs are constructed using plain Smalltalk, as all the components of PHANtom are first class objects. Due to the fact that PHANtom aspects are regular objects, they can be affected by aspects. So it is possible to develop ad-hoc aspect based resolution strategies for aspect interactions, by defining aspects that cut across other aspects.

AspectJ The more widely accepted static aspect oriented language is AspectJ [52]. AspectJ has little built-in support for interactions, mainly aimed at solving precedence issues *i.e.,* advice application order when more than on aspect are applied. The AspectJ Development Tools plugin for Eclipse (AJDT) provides support to visualize the joinpoints captured by an aspect. Therefore, it is left to the programmer to understand if there is some kind of interaction or not. Aspects are also able to cut across other aspects, adding or altering their behavior and structure. Therefore it is possible to implement ad-hoc interaction resolution as additional aspects, as performed by Marot [58], but of course restricted by the limitations of AspectJ's joinpoint model.

Support for Interactions in AOP Languages Even though the support for interaction resolution in preceding approaches seems to be promising, some of them are just a proof of concept (Phase), other are theoretic tools that cannot be applied to the development of an industrial system (Dounce's approach) and others are not maintained anymore (Reflex). Furthermore, as presented in [58], we know that most interactions are very specific. Considering this, it is important to evaluate if the use of general purpose aspect programming languages is adequate to implement interaction resolution.

2.3 Summary

In this chapter we presented aspect orientation concepts, the problem of aspectual interactions and different taxonomies for these interactions. The related work on aspect interactions has been classified according to three phases in the development process that are in the scope of this work, which are requirements analysis, design and implementation phase.

For the design and implementation phases, it can be concluded that existing research work deals with interactions at joinpoint level. We will see in the following chapters that it is a limitation that prevents the use of these approaches in our case study. For the implementation phase, we found that non-formal

AOP languages or frameworks provide limited joinpoint based support for interactions. Consequently, we also regarded AOP languages without advanced built-in support as an alternative to implement ad-hoc interaction resolution logic.

Chapter 3

Slot Machine Domain

As the case study for this work we used *slot machine software*. Beyond our familiarity with it, this software has important features that are relevant to this work:

1. It is a real-world application with a high level of complexity.
2. It presents many crosscutting functional and non-functional concerns.
3. The concerns have not only crosscutting relationships, but they also interfere.

In this chapter we explain some generalities about the slot machine domain, the hardware that is used, the requirements regarding the state that must be stored by the slot machine, connectivity, reporting functionality and how all these features are tested. Finally, we present the concerns found for the slot machine software and the relationships they have.

A slot machine (SM for short) is a gambling device found in casino facilities. It has, usually, five *reels* which spin when a *play* button is pressed. The SM defines many pay-lines: paths whose symbols are matched with certain pre-configured prizes (see Fig. 3.1).

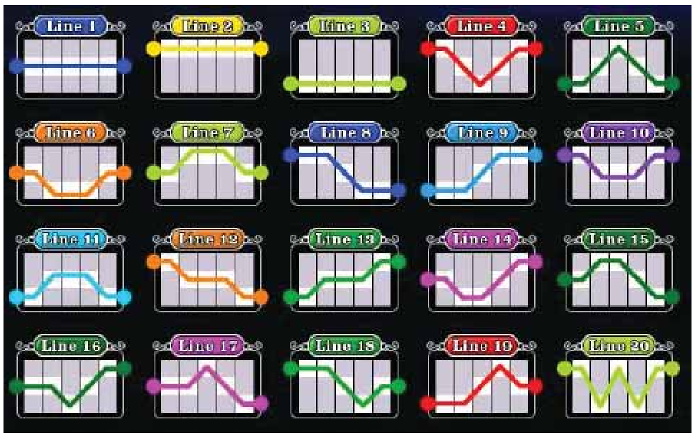


Figure 3.1: Pay lines.

The player selects pay-lines he wants to play, and defines the bet per line. On each play, the SM randomly selects the displayed symbol for each reel, and pays the corresponding prize (if any), according to the *pay-table*.

Bets and prizes are expressed in *credits*. The value of the credit is also configurable. A SM typically accepts payment in bills, coins, or tickets. Once inserted, the money is converted into credits. Once the playing session has finished, the remaining credits can be utilized or "*cash-out*" as coins, tickets or electronic transfers.

3.1 Requirement Sources

The SM game concept is developed by the game designers while its implementation must obey a set of regulations that control both hardware and software. The concept includes the visual look and the probabilistic fundamental behavior. As a game concept can vary from SM to SM, we only focus on the legal regulations, and we furthermore restrict ourselves to regulations for software. These can be divided in three main groups:

Government Regulations Government regulations cover a broad spectrum of characteristics of gambling devices: payout, randomness, connectivity, shared prizes, and so on. One example of these are the Nevada Regulations [63].

Standards To ensure proper behaviour of SMs, there are certification institutes that perform several tests and quality checks on the SMs. The expected behaviour of an SM is defined in documents called standards, for example the GLI standard [40].

Technical Specifications Some requirements are related to the SM's connectivity with *reporting systems* (RS) and the underlying communication protocol. This is the case, for example, of the G2S [41] (Game to Server) protocol, an open standard for communication of an SM with a back-end. There are other propietary protocols that provide similar functionality

Requirements for the SM domain are therefore defined in different documents (regulations, standards, protocol specifications), written by different stakeholders, with diverse interests and backgrounds. We provide more detail and examples of this requirements in Sect. 4.2.

3.2 Rudimentary Design of a Slot Machine

The core design a slot game typically follows is the standard loop pattern depicted in Figure 3.2 [71]. First in this loop, input events are read. These originate either from the user, from hardware drivers, or are scheduled events. Second, the events are processed, which changes the internal game state. Third, the graphical user interface is refreshed to reflect the new status. These operations must be completed in a short time frame. The loop essentially defines the "main thread" of the application. Other actions that are not core to the game

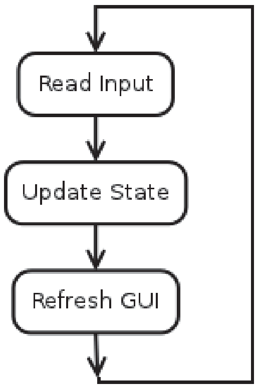


Figure 3.2: Game main loop.

play or that may be time consuming, *e.g.* input/output to external devices, are performed in a parallel thread.

In slot machines, the actions originated by the player include: play, select a bet per line, enable lines, insert money, etc. An example loop is as follows: when the player presses the play button the SM calculates the outcome and, according to the calculated outcome, triggers the reel animation in a separate thread.

3.3 Specific hardware

During their early years, SMs were developed using ad-hoc hardware mechanisms as they were electromechanical (not electronic) devices. They later evolved to become (again ad-hoc) computer based products. In the last twenty years, SM have moved to a standard PC hardware core, allowing the use of mainstream programming languages.

Even though its core is standard PC hardware, the SM must be equipped with several specific hardware items, for example:

- Intrusion sensors: a SM has sensors for detecting intrusion at different levels of the SM cabinet.
- Coin hopper: used to cashout credits using coins.
- Coin Acceptor: accepts coins that are converted into SM credits.
- Bill acceptor: a device for inserting bills into the SM. It only validates pre-configured bills.
- Ticket reader: similar to the bill acceptor, this device reads tickets with an associated credit value. It is common to find a bill acceptor and ticket reader implemented in an all-in-one physical device.
- Thermal ticket Printer: prints tickets to cash out credits from the SM.
- Alarm bell: the bell is sounded to call the attendant.
- Tower lamp: used to call the attendant.

Table 3.1: Devices and some of their events.

Device	**Frequent Events**	**Non-frequent event**
Bill Acceptor	Bill Accepted, Bill Stacked, Bill Returned, Bill Rejected	Bill jam, Bill Stacker Full, Stacker Open
Ticket Printer	Ticket printed	Printer out of paper
Attendant Key switch	Attendant arrived/left	
Sensors	Cabinet Open, Cabinet Closed	CPU enclosure open/closed
dip switches		Reset meters, Demo mode
Coin Hopper	Coin Paid	Hopper Empty
Coin Acceptor	Invalid Coin	Coin cheated

- Attendant key switch: this switch is activated once the attendant inserts a special key in the cabinet. It is used under special circumstances that require human intervention in order to restore the SM to a working state.
- Electro-mechanical meters: these meters count the activity of the machine. They are a kind of view (a copy) of some software meters. Meters are explained in more detail in Sect. 3.4.
- Dip switches: these are hardware switches located in secure places inside the cabinet. They are used, for example, to launch a game in Demo mode or reset meters.

Figure 3.3 shows the cabinet and part of the specific hardware of a SM. The interaction with this specific hardware adds complexity to the SM software as it needs to deal with a lot of device drivers. Special conditions, such as events and errors, need to be appropriately handled or reported to other systems in the casino facility. Table 3.1 presents just some of the events and special conditions raised by the hardware items mentioned.

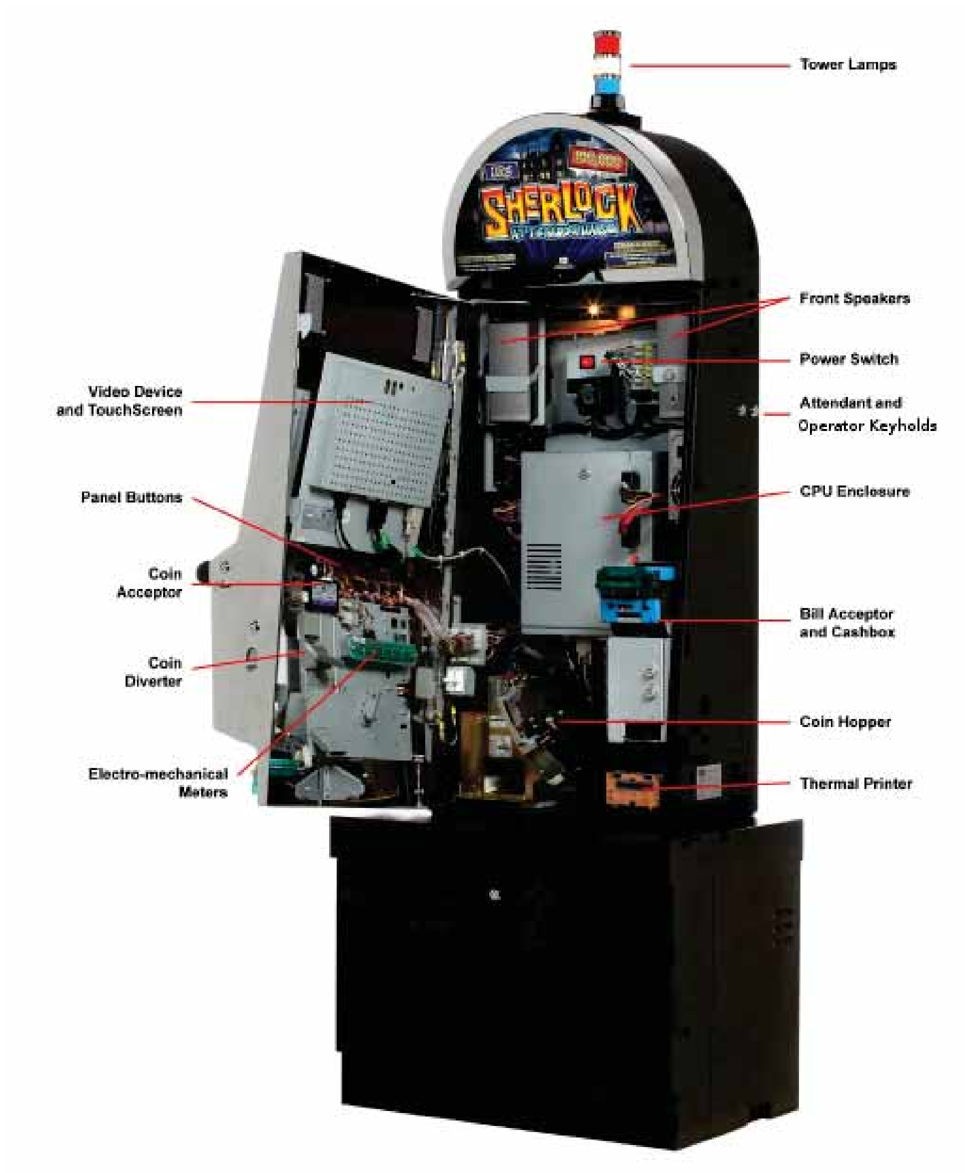


Figure 3.3: Slot machine cabinet

3.4 Meters, Persistence and Recall

SM activity is registered by a large set of counters, called *meters*. There are meters for many purposes and most of them are required by regulations of different countries or states.

Meters are accessed in the SM through a management user interface that allows the attendants and inspectors to read and check their values. Many meters are used to do accounting and share profits between the owner of the machine and the casino. Others are used to measure different aspects of SM performance. There are strict protocols to reset them, and their value must be consistent at any time, as this is highly sensitive information.

There are hundreds of meters, and they cover a wide range of measurable activity: from money in/out, to power cycles of the SM. Table 3.2 presents just a few examples of meters. Progressive refers to the value of the incremental jackpot prize. Tilt counts how many times the SM got locked due to cheating detection. A full list can be derived from different documents including regulations and certification lab recommendations [40]. Note that some meters are calculated based on others, as is the case for Total Drop or Total Out.

Meters and configuration of the SM make up the core information stored in the SM. In the event of a power outage, all the meters need to be recovered to a consistent state. Furthermore, the state of a given play (bet per line, selected lines, outcome if the *play* button was already pressed) must be recovered in order to allow the player to continue with his playing session. This behavior is usually called *Program Resumption* and is required through regulations in almost every jurisdiction.

Regulations also require a feature called *Game Recall*, which provides the ability to inspect the details of last N plays performed in the machine (the lowest number we are aware of is 10). This feature is requested to solve disputes regarding the behavior of the machine. Using this feature, in the event of a player complaint regarding some prize not being correctly paid, the attendant can inspect the history of plays, including bet per line, total bet, enabled lines, and outcome of the game.

3.5 Monitoring

As SMs handle money, they are subject to careful monitoring and accounting. This is needed not only for verification of the correct behavior of the machines, but also for distributing revenues among stakeholders. In order to do almost real-time accounting, SMs are connected to monitoring systems. There are different technologies for connecting them: from serial connections to Ethernet based networks. Different communication protocols are also used. These different communication protocols however provide equivalent functionality, allowing to query the SM state *i.e.,* its meters and configure some aspects of the SMs.

Communication protocols are defined in their corresponding technical specification documents. Some of them work in polling mode, in which the monitoring system polls each machine at a regular intervals. Therefore, the machine can send a message only as a response to a poll. Other protocols are based on web services and both parties can send a message to the other when needed.

Table 3.2: Meter examples.

Topic	Meter	Description
General	Current Credits	Credits available to the player
	Total Drop	Total amount of credits inserted and stored in the SM
	Total Out	Total amount of credits cashed out
	Game Played	Total number of games played since last meter reset
	Game Won	Total number of plays resulting in any winning greater than zero
	Game Lost	Total numbers of plays with no winnings
Paid	Total Jackpot	Total credits paid by the attendant (hand-pays)
	Jackpot	Last credits paid by the attendant (hand-pays)
	Canceled Credits	Total amount of credits cashed out from the SM
	Cumulative Progressive	The sum of all progressive prizes paid
Coin	Coin Drop	Coins diverted to the drop
	Invalid Coins	Invalid coins entered into the SM (this event tilts the SMs)
	Coin In/Out	Physical coins accepted or paid
Bills	Bill In	Total amount (currency) of bill accepted by the validator.
	Items Accepted	Number of bills accepted by the validator
	Bills Per Value	Individual meters counting the number of bills by denomination

In this work we consider the two most widely used protocols. One of them is a proprietary protocol, which prevents us from providing its name. We shall call it SCP. The other protocol is called G2S [41], which is a modern protocol based on web-services. It is open, and its specification can be found on the Gaming Standard Association web page [5].

Communication protocols provide hundreds of types of messages, which can be grouped as follows:

Meters Query Messages These are commands and responses used to query the status of the SM, which is mostly stored in the meters. These meters related messages are used as the main source of information for accounting processes. These messages are sent by the accounting backend system.

General SM Configuration SMs can be configured remotely using the communication protocols. For instance, it is possible to set the current time, or select the desired payout profile of a SM. In multi-game machines, it is also possible to select which game should be displayed.

Ticket-in and Ticket-out Messages In some casino facilities, cash is not used. Instead, when the player arrives, he buys a certain amount of credits in the form of a ticket. The ticket only contains an identifier and not a monetary value. When inserted in a SM, the corresponding credits are "transferred" to the SM. Once the player decides to cashout, a new ticket is issued by the SM. The information regarding valid tickets and their credit value is stored in centralized servers.

Real Time Event Reporting Events during the game cycle (BeginGame, EndGame) must be reported to the monitoring systems. Events and error conditions described in Sect. 3.3 should be reported when possible. This conditional reporting feature is due to the fact that not all the devices allow the detection of exactly the same events. In other cases, event reporting, for certain types of evens is not required by regulations, though useful for casino management tasks.

A SM can be connected to more than one remote backend at the same time. Configuration is a very sensitive part of these protocols. Configuration inconsistencies may result in the SM misbehaving and paying incorrectly.

3.6 Certification and Demo mode

The features mentioned so far are usually requested through regulations imposed by the jurisdiction where the SMs are deployed. These features must be present in every game before it arrives to the market. In order to ensure that the games comply with the requirements, there are institutions devoted to the *certification process*. These are called certification labs and are in charge of verifying SM performance on a wide range of aspects, from electrical to game logic, randomness and communication.

Certification is a laborious process where many aspects from the game are compared with their *expected* behavior. We will not describe the electrical and electronic part of certifications, as it is out of the scope of this work. From the software point of view, there are many elements analyzed during this process. The most relevant are:

Random Number Generator The random outcome must be generated by an approved random number generation algorithm. This ensures certain properties on the probabilistic distribution of the results.

Payout The game must behave in correlation to the reporting payout for each configuration. That is, in a long run (a million plays at least) it should return to the player a certain percentage which depends on the pay table and symbol configuration.

Pay-table payment The pay-table defines a map from symbol combinations to prizes. The game is checked to ensure it pays the correct amount of credit for each symbol combination declared in the pay-table.

The pay-table testing requires that all configured prizes be awarded so that their payment can be checked. Big prizes have less probability of being awarded so playing manually until a big prize is awarded is not feasible. To solve this problem, certification labs require the SM to provide a special running mode, called Demo. Demo mode provides a way of advising the SM to pay a certain prize the next play. In this way, the certifier can select the different prizes of the pay-table and check the correct payment of them. Of course, the demo mode switch is out of reach of the player. In fact, it is implemented as a dip switch, inside the CPU enclosure.

3.7 Concerns in Slots Machines

Considering our intention of applying aspect orientation to the SM domain and based on our experience in the domain and the set of legal requirements that apply, we organize the requirements in the following concerns:

Game This is the basic logic of a gambling device. The user can enter credits into the machine, and then play. The output is determined randomly and when the player wins, he is awarded an amount of credits.

Meters This refers to the set of counters that are used to audit the activity of the game. Recall that for instance, there are meters that count the number of plays, the total amount bet, total won, error condition occurrences, and so on.

Program Interruption and Resumption Program Resumption is a persistence and recovery set of requirements. Requirements in this concern determine how the machine should behave after a power outage, specifying which data need to be recovered. The system should recover the last state or setting after a power outage.

Game Recall This refers to the information that must be available about the current and previous plays, in order to solve any dispute with players.

Error Conditions Under certain circumstances, the game should detect error conditions and behave accordingly. This concern defines what are considered error conditions and how the game must react to them.

Communications The SM is connected to a monitoring system in the casino. This concern defines the kinds of data, the format and when data must be exchanged between the SM and the monitoring system. Recall that several communication protocols exist, each with their own specification that states what data needs to be persistent, which meters are necessary, and so on. For our work, we consider either using G2S or SCP. In the remainder of this thesis, we will refer to the generic *Communication Protocol* concern or, when appropriate, we will specify which protocol we are considering.

Demo The demo concern contains the requirements specifying how the game behaves in this mode. Playing the game in demo mode makes it is possible simulate events such as entering money or winning a prize. Note that any meter or data changed due to operation in demo mode should not be stored or reported, as it is simulated behavior.

These concerns match the definition of Brito *et al.* for concerns in AORE [15], which must refer to a coherent set of requirements that allude to a property or feature that the system must provide. Some other concerns, such as the game story line and bonus rounds, haven omitted in order to keep the discussion focused. As the domain is complex, there may be other possible concern decompositions. We choose this one because, based on our experience, it properly modularizes the different required features of slot machines and shows the interactions that are at the core of this work. We expect some of the selected concerns to become components and others aspects. Different instantiations of the SM software may include different implementations or compositions of components or aspects to comply with the regulations of each scenario.

Recall that an aspect adds its behavior at a joinpoint (captured by a pointcut). Examples of interesting joinpoints in the SM, include the *play* of a game or the execution point where error conditions are created. At these execution points aspect must execute their behavior.

3.8 Interactions

As mentioned in Sect. 2.2, we classified interactions using the taxonomy described by Sanen *et al.* (conflict, dependency, reinforcement and mutex). Based on our previous experience implementing software for SMs, we observed the existence of these aspect interactions between the concerns identified in our domain. For example, there is a conflict between the *Demo* and *Meters* concerns, since *Meters* works correctly without *Demo*, but if *Demo* mode is active, activity in the machine must not be counted by *Meters*. An example of mutex is in the communication protocols: it is forbidden to have two protocols providing the same functionality at the same time. A dependency example is the relationship between *Communications* and *Meters*; the protocol needs to communicate the status of the SM, which is in part represented by the meters. Finally, a reinforcement is present between *Error Conditions* and *Communications*. The existence of error condition detection enables communication protocols to provide "extra" functionality, in this case real time error condition reporting.

Understanding how concerns interact with each other is key information that needs to be passed to designers and programmers. For example, in the case of a dependency, the dependent concern will be affected by design decisions upon the other. On the other hand, if there is a mutex relationship, architectural mechanisms should be provided to ensure that both concerns (or parts of them) will not be active at the same time.

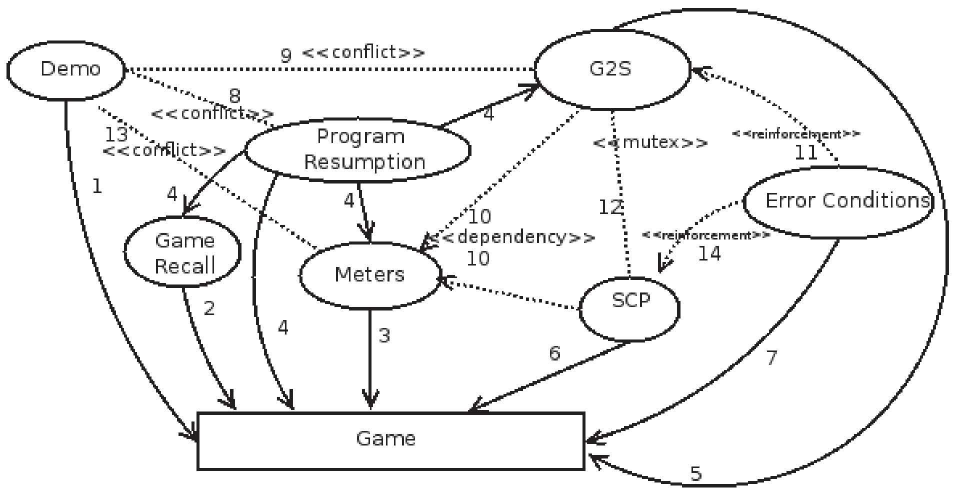


Figure 3.4: Concern interactions in an ad-hoc notation. Regular arrows indicate crosscutting, dashed arrows indicate interactions between concerns, tagged with UML-like stereotypes.

Considering the concern division and the associated requirements, we have deduced the relationships between different concerns and identified their interactions. A representative selection of these is shown in Fig. 3.4, which uses an ad-hoc notation where *Game* is the base concern (represented by a square) and crosscutting concerns are represented by ovals. Different arrows are used to indicate crosscutting and interaction relationships. More in detail, the **crosscutting** relationships are as follows:

1. *Demo* to *Game*: The demo requirements affect many of the definitions of the original requirements of *Game* in order to alter the *Game*'s behavior for testing purposes.

2. *Game Recall* to *Game*: *Game Recall* requirements affect many aspects of the *Game*'s behavior, it is used to log the activity for each play and other relevant events in order to solve disputes.

3. *Meters* to *Game*: *Meters* count activities of many functions defined in *Game*, for instance: game play, bill in, cashout, etc.

4. *Program resumption* to *Game*, *Game Recall*, *G2S* and *Meters*: *Program resumption* is analogous to persistence. It crosscuts all the places where important data, which needs to be restored, is changed.

5. *G2S* to *Game*: This concern cuts across many *Game* requirements, since several events in Game need to be reported, monitored and communicated to the reporting system.

6. *SCP* to *Game*: This refers to the other protocol used to monitor the game's behavior.

7. *Error Conditions* to *Game*: The behavior associated with error conditions need to be woven into the game behavior. Requirements in *Game* that could raise an error condition vary: from a bill jam to a door opened.

The are many **interactions** between these concerns. We highlight the the following:

8. **Conflict** between *Demo* and *Program Resumption*: The demo mode fires *fake* events that must not be counted nor restored after program interruption.

9. **Conflict** between *Demo* and *G2S*: both concerns cannot be active at the same time because demo fires events that must not be reported to the monitoring system. The same conflict exists forbidden Demo and SCP, it is not added in the diagram to avoid clutter.

10. **Dependency** of *G2S* and *SCP* on *Meters*: Some data reported to the monitoring system is stored or can be derived from meters. Hence, communication protocols need the meters to be up to date in order to accomplish their purpose.

11. **Reinforcement** of *G2S* with *Error Conditions*: As we discuss in Sect. 5.2.2, some parts of the G2S protocol are not mandatory for specific instances. When error conditions are tracked in the game, additional behavior is made available in G2S such as particular parts of real time event reporting functionality.

12. **Mutex** between *G2S* and *SCP*: There is overlapping functionality defined in the requirements of both protocols. Recall that backends can configure the SM remotely (see Sect. 3.5). For example, both of them are used to keep the time in sync between the SM and the monitoring system. Having both protocols active, with monitoring systems out of sync, would render the time of the SM inconsistent. Therefore they cannot be active at the same time.

13. **Conflict** between *Demo* and *Meters*: Meters must not be affected by activity in Demo mode.

14. **Reinforcement** of *SCP* with *Error Conditions*: it is similar to 11, additional behavior in SCP is available when new error conditions are supported.

This is only a selection of some representative interactions. There are many more that we do not include in Fig. 3.4 nor talk about in detail. For example: G2S and SCP depend on GameRecall as both protocols require to retrieve information of the last plays, and Program Resumption crosscuts SCP for the same reason as it crosscuts G2S. We effectively selected just one case of each interaction type to keep the discussion focused.

3.9 Summary

In this chapter we have succinctly described the relevant characteristics of the domain in order to illustrate the intrinsic complexity of this kind of software. The concerns found on this domain are:

- Game
- Meters
- SCP Communication protocol
- G2S Communication protocol
- Game Recall
- Program Resumption
- Demo
- ErrorConditions

Some of them are base concerns and others are crosscutting. These concerns present a number of interactions which can be classified according to Sanen *et al.*. [74] into: mutex, reinforcement, dependency and conflict. We selected the following interaction instances for this work:

- Conflict between Demo and Meters, Game Recall, Program Resumption and Communication protocols
- Dependency of Communication Protocols (SCP and G2S) on Meters.
- Reinforcement of Error Condition on SCP and G2S communication protocols.
- Mutex between configuration features SCP and G2S communication protocols.

These interactions have considerable impact on the game behavior. In our experience not having proper mechanisms for handling them has been the source of costly bugs in the past. Therefore, in the remainder of this work, we analyze how these interactions can be handled in each stage of the development cycle and how information about them can be traced along the development process.

Chapter 4

Interactions in Analysis

> This chapter is based on our previously published work *Expressing aspectual interactions in requirements engineering: Experiences, problems and solutions* [97].

The first step in software development consists of understanding what the software should do. The input for this are system requirements. Requirement engineering is the process that produces the initial documents and models which are used as the basis for design and implementation. Therefore, we need to start modeling with requirements, concerns and their interactions.

This chapter presents requirement engineering with emphasis on the modeling of aspect interactions. The following section provides details on the requirements for SMs, and how two specific AORE approaches support the peculiarities of the SM requirements. In addition, it also presents the extensions we made to these approaches in order to handle SM requirements effectively and the experiments performed to validate their applicability.

The chapter is organized as follows: Sect. 4.1 presents an overview of AORE from the aspect dependencies and interactions perspective. Sect. 4.2 goes in more detail on the slot machine requirements and some relevant characteristics of their sources. The evaluated AORE approaches and the evaluation output are presented in Sect. 4.3, including the limitations found for each approach. In Sect. 4.4 we propose extensions to the original approaches in order to cope with the limitations found. Sect. 4.5 describes a user study carried out in order to validate the expressiveness of the extensions performed to MDSOCRE. Sect. 4.6 concludes the chapter. Two appendixes, (A and B), at the end of this thesis contain the complete models we produced for this stage, showing the results of applying the extensions to our setting.

4.1 Requirements Engineering and Aspect Dependencies and Interactions

In the requirements engineering community, interactions between requirements are a well-known (intra) traceability problem [67]. In the AOSD community, and hence when applying AORE techniques, this problem maps to what is known as dependencies and interactions with aspects [17]. Dependency and interaction support is still an open issue for the AOSD community. Therefore, the focus of our work lies in assessing the support for expressing interactions between aspects or concerns, which, depending on the point of view, can be considered either a requirements traceability problem or a problem of dependencies and interactions with aspects.

Non-functional requirements approaches support conflict resolution, providing algorithms to find the best trade-off for the stakeholders [24]. In these approaches, some requirements are **not** completely satisfied, in favor of other more relevant ones (according to contribution analysis). In our domain however, all of the requirements need to be fulfilled (none can be discarded), and their fulfillment is verified during the certification process (see Sect. 3.6). Instead of removing requirements, we need to deal with conflicts by not allowing the activation or execution of conflicting features during a single system run. In addition to this, it is interesting to note that most of our interacting crosscutting concerns are **functional** ones. In conclusion, typical non functional requirements (NFR) approaches are not applicable in our scenario.

We elected to perform requirements engineering using two of the approaches presented in Sect. 2.2.1: The Theme/Doc approach [9] and the Multidimensional Separation of Concerns for Requirements Engineering (MDSOCRE) approach [60], focusing on how these allow us to express and document aspectual dependencies and interactions. The choice of these two approaches was based on our perception of their maturity and of their acceptance in the AORE community.

More concretely, Theme/Doc was selected, not only by its publication record, but also because a book is available that describes at a detailed level its application to a arguably complex example (which also demonstrates some degree of scalability) [26]. Moreover, it also discusses later phases of development, for example how to deal with conflicts in design documents when using Theme/UML. MDSOCRE was selected because it claims to explicitly support conflict resolution and because of its flexibility due to the action/operator approach (discussed in more detail in Sect. 4.3.4) that allows us to express more than just crosscutting relationships.

Both of the approaches we evaluated enable us to partially express the requirements, but neither of them was entirely satisfactory. Theme/Doc lacks support for the kind of interactions we want to model, *e.g.* conflicts between aspects, and needs a finer granularity for expressing crosscutting relationships. MDSOCRE lacks explicit support for expressing interactions between concerns when they do *not* cross-cut each other. In this chapter, in addition to describing these drawbacks in more details, we also elaborate and validate extensions that address these issues. We propose new kinds of relationships to Theme/Doc for documenting interactions, and add new interaction information in MDSOCRE.

4.2 Requirements in the Slot Machines Domain

As said in Sect. 3.1, the SM game concept is developed by the game designers while its implementation must obey a set of regulations that control both hardware and software which can be divided in three groups (described in Sect. 3.1):

Government Regulations that specify the accepted behavior of the SM regarding: payout, randomness, shared prizes, and so on. For example the

Nevada Regulations [63].

Standards Which are defined by certification laboratories. They specify the expected behavior of an SM under certain circumstances for example, in which situations the SM must call the attendant for a hand pay. This are usually complementary with the government regulations.

Technical Specifications These requirements establish requirements regarding connectivity with the *reporting and monitoring systems* (RS). They define fundamentally the underlying communication protocols, for example the G2S [41] (Game to Server) protocol.

These requirements are contained in documents, written by stakeholders with different backgrounds and interests. Furthermore, this results in a sizable set of documents using multiple terms for describing the same object, action or event. We count approximately 150 pages of pure requirement specifications, while the complete documents that include clarifying text, examples and notes are approximately 2000 pages in size. Furthermore, in some cases it is necessary to complement and normalize different sources referring to the same topic. For instance, consider the case of *Error Conditions*, which are treated by both the Nevada regulations [63] and the GLI standard [40]; some of the conditions specified by the regulations match, but others are defined by only one of them. Note that the final SM, if needs to be installed in Nevada, must comply with the error conditions defined in both documents.

Lastly, an important characteristic of communication protocol requirements (e.g. in G2S) is that they are divided in *topics* and that not all the topics are required for certain deployments. As a result, part of the communication functionality is optional, which has an impact on requirements modeling.

4.2.1 Selected Requirements

Due to the large number of requirements in our case (around 600), we only show here a small subset of requirements that illustrate the main concerns for the SM domain and the four types of interactions described by Sanen *et al.* [74]: conflict, dependency, mutex and reinforcement.

The requirements we present here are slightly simplified and summarized versions taken from the different original sources (regulations, standards and protocol specifications). These documents are produced by the corresponding organizations: certification laboratories, protocol steering committees, and institutions that regulate the gaming industry. It is important to note the particular nature of these requirements: as these are legal requirements, **all** of them must be fulfilled in the resulting application. Moreover, all of them are known and presented at the start of the development process. Part of the formal certification process of the software that is installed in the SMs is ensuring that all these legal requirements are met. Also, the requirements we discuss here change at a slow pace (which can be measured in years) when compared with the time taken by the development of a SM game. This is due to the nature of the source organizations that produce the legal requirement documents.

We have grouped the requirements according to the concerns presented in Sect. 3.7 to favor the understanding of the different interactions which are the core of this work. In Tables 4.1 and 4.2, we summarize some requirements, grouped by concern and not by the document where they are defined. For a complete listing of them, we refer to the various requirements documents [40, 41, 63].

4.3 Evaluation of AORE Approaches

From the approaches presented in the related work (Sect. 2.2.1), we selected Theme/Doc and MDSOCRE in order to evaluate their applicability. For both of them we based our selection on their publication record and acceptance in the AORE research community. In the case of MDSOCRE , we also evaluated that it is part of a line of AORE approaches, which also have a measure of flexibility (relationship among concerns can be configured). In the case of Theme/Doc, it is important for us that it has a book with detailed description of the notation and methodology and that it is integrated with a design approach (Theme/UML). We provide more details of these approaches in the following sections.

Table 4.1: Requirements for SM domain concerns.

Game			
R-SM-1	Slot machines have 5 reels.	R-SM-4	A slot machine has one or more devices for entering money.
R-SM-2	Reels spin when play button is pressed.	R-SM-5	As money is inserted credits are “assigned” to the player.
R-SM-3	Prizes are awarded according to a pay table.	R-SM-6	A slot machine must provide some means for cashing the credits out. It could be a ticket printer, a coin hopper, etc.
Game Recall			
R-GR-1	Information on at least the last ten (10) games is to be always retrievable on the operation of a suitable external key-switch, or another secure method that is not available to the player.	R-GR-2	Last play information shall provide all information required to fully reconstruct the last ten (10) plays. All values shall be displayed; including the initial credits, credits bet, and credits won, pay-line symbol combinations and credits paid whether the outcome resulted in a win or loss [...]. This information should include the final game outcome, including all player choices and bonus features.
Program Interruption and Resumption			
R-PR-1	After a program interruption (e.g. processor reset), the software shall be able to recover to the state it was in immediately prior to the interruption occurring.	R-PR-3	An SM must store all meter information in persistent memory.
R-PR-2	Restoring Power. If a gaming device is powered down while in an error condition, then upon restoring power, the specific error message shall still be displayed and the gaming device shall remain locked-up [...].		
Demo Mode			
R-DM-1	The Slot Machine must permit a test, diagnostic or demo mode, which permits a gaming device (*e.g.* a hopper) to be tested. The test shall be completed on resumption of normal operation.	R-DM-5	Specific meters are permissible for these types of modes provided the meters indicate as such.
R-DM-2	If the gaming device is in a test, diagnostic or demo mode, any test that incorporates credits entering or leaving the gaming device (e.g., a hopper test) shall be completed on resumption of normal operation.	R-DM-6	Test/diagnostics mode may be entered, via an appropriate instruction, from an attendant during an audit mode access. These modes should not be accessible to the player.
R-DM-3	There shall not be any mode other than normal operation (ready for play) that increments any of the electronic meters.	R-DM-7	When exiting from test-diagnostic mode, the game shall return to the original state it was in when the test mode was entered.
R-DM-4	Any credits on the gaming device that were added during the test, diagnostic or demo mode shall be automatically cleared before the mode is exited.	R-DM-8	Test Games. If the device is in a game test mode, the machine shall clearly indicate that it is in a test mode, not normal play.

Table 4.2: Requirements for SM domain concerns (continuation).

Meters			
R-M-1	Credit meter: shall at all times indicate all credits or cash available for the player to wager or cashout.	R-M-3	Accounting Meters: *Coin In*: [...] a meter that accumulates the total value of all wagers [...]. *Games-played*: accumulates the number of games played; since power reset, since door close and since game initialization.
R-M-2	Credit Meter Increment: The value of every prize (at the end of a game) shall be added to the player's credit meter [....]. The credit meter shall also increment with the value of all valid coins, tokens, bills, Ticket/Vouchers, coupons or other approved notes accepted.	R-M-4	Meters should be updated upon occurrence of any event that must be counted, including: play, cashout, bill in, coin in.
R-M-5	G2S meters are: *gamesSinceInitCn* Number of games since initialization. *WonCnt*: Number of primary games won by the player. *LostCnt*: Number of primary games lost by the player.		
Communications: G2S			
R-G2S-1	The G2S protocol is designed to communicate information between an SM, and one or more host systems.	R-G2S-4	The device can generate an event in a unsolicited manner or in response to a host command
R-G2S-2	Meter information can be queries by a host in real-time or a host may set a periodic subscription to cause the SM to send selected meters at predetermined intervals.	R-G2S-5	Current time-stamp can be set by the host.
R-G2S-3	Information provided by the SM is used for audit purposes.	R-G2S-6	Command GetGameRecallLog is used by a host to request the contents of a transaction log of last plays from a SM.
Communications: Proprietary Comm. Protocol (SCP)			
R-SCP-1	The SCP communicates a SM with a host system.	R-SCP-2	The SCP must report meters of a SM when required by the host.
R-SCP-3	Configuration settings such as current time-stamp are configured from the host.		
Error Conditions			
R-EC-1	Gaming devices shall be capable of detecting and displaying error conditions and illuminate the tower light for each or sound an audible alarm.	R-EC-5	Error conditions shall be communicated to an on-line monitoring and control system when this is available.
R-EC-2	Error conditions should cause the gaming device to lock up and require attendant intervention. Error conditions shall be cleared either by an attendant or upon initiation of a new play sequence after the error has cleared except for those deemed as a critical error.	R-EC-6	An event represents an occurrence of an incident detected by a device in an EGM.
R-EC-3	Error conditions are: coin jam, reverse coin in, stacker full, bill jam, external doors open.	R-EC-7	Important events must be reported in real-time, including: error conditions, tickets inserted, ticket printed.
R-EC-4	Video based games shall display meaningful text as to the error conditions.		

4.3.1 Theme/Doc

Theme/Doc is an AORE methodology that, apart from being mature and accepted in the AORE community, is part of a more comprehensive approach called Theme [9, 26], which also treats aspectual design (Theme/UML).

We selected Theme/Doc because it explicitly supports passing information from the requirements analysis to the design phase, which could include interactions. Besides this, the Theme book [26] mentions conflict resolution as a feature of the Theme approach, which is one interaction type we are interested in. Also, in terms of scalability, it was applied to a non-trivial example in the book, which makes it a candidate for our evaluation.

Brief overview of Theme/Doc

Theme/Doc [8] is the requirement analysis part of the Theme approach [9, 26]. In Theme/Doc, requirements are organized into concerns, called *themes*. Themes can be defined through an initial set of domain specific actions or concepts, others may be recurring typical concerns: persistence, logging, and so on.

In Theme/Doc a requirement is attached to a theme if the name of the theme appears in the requirement. In other words, Theme/Doc relies on the name-based analysis of actions in requirements to relate them to themes. In our study, we did not strictly follow this rule. Instead, we use the concerns we identified in Sect. 3.7 as themes. We will detail our motivation for this in Sect. 4.3.3.

Ideally, each requirement should belong to one theme, but chances are that some of them are shared among themes, *i.e.,* crosscutting. In Theme/Doc, a shared requirement is considered crosscutting if all of the following four conditions are satisfied [26]:

1. The requirement cannot be split in order to avoid tangling.
2. One of the themes dominates the requirement: it has a stronger belonging relationship with one of the themes.
3. The dominant theme is triggered by events in the base theme: the behavior described by the dominant theme is fired as a result of the execution of some behavior from the base theme.
4. The triggered theme is fired in multiple situations: the crosscutting behavior must be executed in several cases, not just one.

A crosscutting theme is one that contains at least one crosscutting requirement. An example of the application of the mentioned rules is as follows: requirement R-M-4 from Meters mentions that plays and other events must be counted. The base theme is Game, where the plays and other relevant events occur. The dominant theme is Meters, whose requirement (count events) is fired by events in the base. Therefore, Meters is a crosscutting theme.

An important feature of Theme/Doc is its visual support through diagrams, which helps in the understanding of the requirements model of the system. In Theme/Doc diagrams, requirements are represented by rounded boxes, and

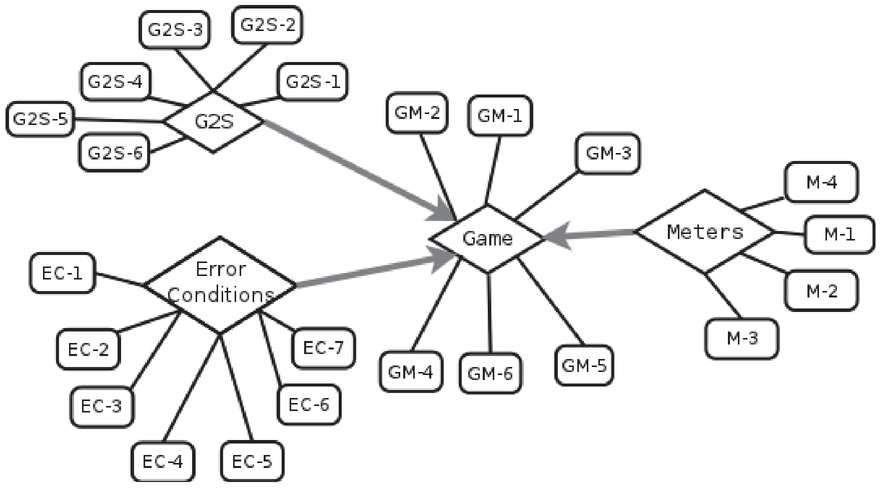


Figure 4.1: *Game*, *Meters*, *G2S* and *Error Conditions* concerns expressed using the Theme/Doc notation.

they are organized around themes, which are depicted by diamonds. When a crosscutting theme exists, a gray arrow is drawn from the theme that crosscuts, *i.e.,* the aspect, to the theme that is being cut across, *i.e.,* the base. Consider for example Fig. 4.1, where *Game*, *Meters*, *Error Conditions* and *G2S* concerns are represented along with their requirements and crosscutting relationships. Unfortunately, as we report in the following sections, it is not possible to document which requirements participate in the crosscutting relationships.

The Theme book [26] claims that Theme/Doc has tool support through a web application[1]. Unfortunately, this software was not available at the time of our experiments. To the best of our knowledge, there are no other tools with direct support for Theme/Doc. We therefore followed the notation from the book to build our diagrams by hand.

4.3.2 Use of Theme/Doc

As shown in Fig. 4.1, the graphical approach of Theme/Doc makes it easy to read the relationships between requirements and themes. Each theme can be easily identified along with its associated requirements. The four steps to check for crosscutting helped us to correctly establish which were the crosscutting concerns. In the resulting diagrams, the crosscutting relationships are clear, enabling us to easily identify which concern is playing the base and/or the aspectual role. However, the involved requirement in these relationships cannot be expressed using Theme/Doc.

Figure 4.2 shows crosscutting among the themes presented in Sect. 3.8. For clarity, here we just present the crosscutting relationships between the themes without including the requirements. The diagrams showing all the above concerns with their more significant requirements are included in A. Note that in contrast with Fig. 3.4 (which uses an ad-hoc notation for showing crosscutting and interaction relationships) in this case only the crosscutting information is

[1]Stated to be available at: http://www.thethemeapproach.com:8081

present due to the limitations of Theme/Doc, which are analyzed in the following sections.

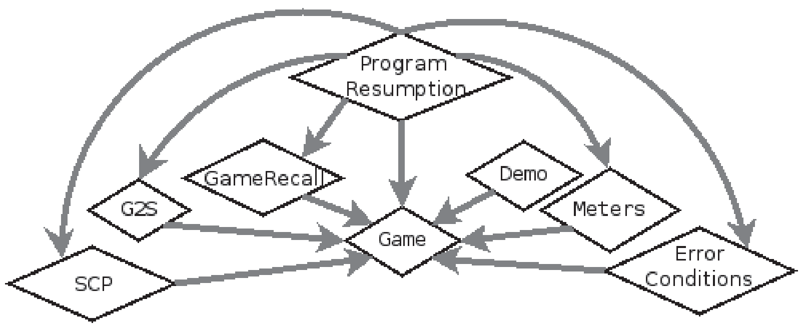


Figure 4.2: Crosscutting relationships between themes using the Theme/Doc graphical notation.

4.3.3 Limitations of Theme/Doc

In our evaluation, we encountered the following limitations of Theme/Doc.

Granularity

As explained before, gray arrows denote crosscutting. As each concern potentially contains many requirements, it is difficult to discern which specific requirement of the crosscutting theme affects which requirements on the base theme. Consider for example Fig. 4.1 and the crosscutting relationship between *Meters* and *Game*; here it is not possible to know which requirement in *Meters* is crosscutting. Furthermore, it is not possible to know which specific requirements in *Game* are affected as the result of the crosscutting. Where possible, it is desirable to pass that information to the design phase, so that base and aspectual components can be properly designed. In fact, this information is available during the analysis phase – in the identification of crosscutting themes – of Theme/Doc, but it is not made explicit.

Expressing Interactions

In Fig. 3.4 we show different examples of interactions between aspectual concerns for requirements. If we consider Fig. 4.1, we can however see that these interactions are missing. This is because Theme/Doc lacks support for expressing interactions. For instance, missing in Fig. 4.1 is a dependency of *G2S* on *Meters*. This information is however crucial: Multiple perspectives of a system (*themes* in this case) need to be combined to form a system [85]. We require the dependency information to select a sound set of themes for a system. For example, it is not possible to build an SM with *G2S* support, but lacking *Meters*. This is because *G2S* requires the existence of *Meters* to provide its own functionality. The same applies to *SCP* and *Meters*. The same happens with *conflicts*, for instance, between *Demo* and *Meters*. It is critical to know that architectural or design mechanisms need to be included to avoid the activation of both concerns at the same time. Developing the system without this information would entail costly fixes in the future, when the interaction is encountered. The reinforcement from *Error Conditions* to *G2S* is also missing. Documenting it signals that an optional part of G2S is active when *Error Conditions* are available.

Implicit Requirements

As a consequence of performing a domain knowledge based analysis of our requirements and concerns we also were confronted with the fact that the information contained in the original requirements, *in some cases*, needs to be combined with domain knowledge. This is needed in order to generate new requirements that are more suitable for understanding concern relationships. It is similar to the approach proposed by Bar-On et al. [11], where implied actions are used to generate new *derived requirements*.

For example, during interaction analysis, it is possible that new requirements arise because of interactions that need to be resolved. Consider the following example:

> G2S provides time and date configuration for slot machines. SCP also provides the same configuration. A SM can be connected using both protocols at the same time.

In this case there is an implicit mutex relationship present. The machine can take the time and date from any of the protocols, but not from both of them at the same time. This would lead to erratic behavior in case that both times do not exactly match. Hence, the following issue arises: Which time and date source should the SM take when both protocols are active? A decision must be made in the requirements analysis, for example, stating that G2S is preferable over any other source of time and date. This decision helps to resolve the interaction problem by defining part of the behavior of the system, and must be recorded.

Ambiguity of the Requirements

An ideal requirements specification should be complete, unambiguous, verifiable, consistent, modifiable and traceable [1]. Under these assumptions Theme/Doc should work smoothly. Unfortunately, in our particular case there is no single requirements specification unifying all the sources and we are faced with significant ambiguity. The variety of sources first results in synonyms being used in different documents. Second, and more importantly, there are complete key ideas, concepts or interactions that are expressed using different vocabulary and style. Although we might consider our case as being exceptional, we consider it worthwhile to examine the impact this has on Theme/Doc.

The ambiguity we face affects the mechanism proposed in Theme/Doc to assign requirements to themes, and to identify potential crosscutting themes. For example, consider the case of attaching requirements to themes, where it is necessary to look for a theme's name in the requirements. In our case, sometimes the theme's name is represented by a phrase or an adjective, which gives the analyst an indication to attach it to the theme. In the worst case however, the requirement and the theme could be related by implicit actions, as outlined above. Furthermore, the same issue is present when crosscutting relationships are identified. According to Theme/Doc, shared requirements are potential indicators of crosscutting. Having a shared requirement means that two concerns are present in the text [26]. This suffers from the same drawback of requiring unambiguity.

Baniassad and Clarke [9] have shown how Theme/Doc analysis of actions helps to solve some ambiguities and how a synonym dictionary helps in the case of multiple terms referring to the same concept. In our experience, the problem goes deeper than the use of synonyms: we not only have some words that are written in different way, sometimes ideas are equivalent but explained differently. Put differently, the name-based approach proposed originally by Theme does not work in our setting due to the nature of the requirements sources. As an example, consider the case of requirements R-PR-3 and R-DM-7 (from Tables 4.1 and 4.2. Note that these requirements refer to the same concepts, but textual analysis fails due to the impossibility of matching words. Having several large documents which present not only words but key ideas using a different vocabulary prevents the use of the name-based approach of Theme/Doc.

Considering the kind of requirements we face, we consider two options to resolve ambiguities. The first one is to rewrite all the requirements, normalizing them to use the same vocabulary; the second one is to use domain knowledge to associate requirements with the corresponding themes directly. Due to the large number of requirements (approximately 600) and presence of multiple sources, the first option is not feasible, we therefore opt for the second. Note that grouping requirements into concerns based on domain knowledge is not new [60, 10]. Our experience is that the resulting concerns are useful as they can be easily discussed with domain experts.

In conclusion, Theme/Doc lacks support for the identification and notation of derived requirements

4.3.4 MDSOCRE

MDSOCRE (Multidimensional Separation of Concerns in Requirements Engineering) is the evolution of a line of AORE approaches such as PreView [77] and ARCaDe [70]. As it arguably provides the most expressive and flexible constructs for binding crosscutting concerns to base concerns, we chose it as the second case in our study.

Brief Overview of MDSOCRE

Multidimensional Separation of Concerns in Requirement Engineering (MDSOCRE) [60], is a refinement of the ARCaDe approach [70]. MDSOCRE treats the concerns in a uniform fashion, regardless of the nature of the requirement (functional or non-functional). It makes it possible for the requirement engineer to choose a subset of requirements to observe the influences on each other and to analyze crosscutting behavior. In contrast to Theme, it does not provide a graphic notation, instead it uses XML to express requirements and composition rules.

MDSOCRE aims to eliminate conflicts, which are the result of contradictory concerns. These are detected and handled using contribution matrices. In such a matrix, rows and columns identify concerns and the cells denote how the concerns contribute to one another (negative contributions denote conflicts). These matrices help in the decision process of which (parts of) features will be implemented. Conflicts in MDSOCRE differ from our definition in Sect. 3.8. In our case, concerns are not a subject of negotiation, as all are required by some standard or regulation. We must however check that at run-time conflicting concerns are not simultaneously active.

MDSOCRE also provides support for *meta concerns*: generic concerns that are instantiated for specific systems. The most important feature of meta concerns for us is their capability for expressing commonly related concerns. We will use this to express interactions in Sect. 4.3.6.

MDSOCRE is supported by the EA-Miner tool [73]. This tool is an Eclipse plugin[2] that needs the requirements to be entered in plain text format. As part of its processing, EA-Miner uses a web service to parse natural language. Unfortunately, this service was not working at the time of our experiments, and we were not able to use the tool.

4.3.5 Use of MDSOCRE

We were able to build a complete requirements model of the SM application using MDSOCRE. Listing 4.1 shows how some of the concrete concerns of our domain are expressed in this approach. The `Concern` tag is composed of several requirements which are surrounded by the `Requirement` tag. A requirement can be referenced by its identifier (*id*) and can contain nested sub-requirements. We do not include the detailed listing of all concerns here and instead refer to the Appendix B

Listing 4.1: Game and Meter concerns expressed using MDSOCRE.

```
<Concern name="Game">
  <Requirement id="1"> A slot machines have 5 reels.
  </Requirement>
  <Requirement id="2"> Reels spin when play button is
   pressed.</Requirement>
  <Requirement id="3"> Prizes are awarded according to a pay table.
      </Requirement>
  <Requirement id="4"> A slot machine has  one or more devices for
      accepting money.</Requirement>
  <Requirement id="5"> As money is inserted credits are "assigned"
      to the player. </Requirement>
  <Requirement id="6"> A slot machine must provide a means for
      cashing the credits out. It could be a ticket printer or coin
      hopper.
</Requirement>
</Concern>

<Concern name="Meters">
   <Requirement id="1"> Credit meter: shall at all times indicate
       all credits or cash
          available for the player to wager or cashout  (GLI 11
              4.10.1)
  </Requirement>
  <Requirement id="2"> Credit Meter Incrementing: The value of every
       prize (at the end of a game) shall be added to the player’s
```

[2]Downloadable from http://www.aosd-europe.net/deliverables/d108EAMinerVersion2.zip.

```
        credit meter. The credit meter shall also increment with the
        value of all valid coins, tokens, bills, ticket/vouchers,
        coupons or other approved notes accepted. (GLI 11 4.10.5)
  </Requirement>
  <Requirement id="3"> Accounting Meters (GLI 11 4.10.9):  Coin In:
        a meter that accumulates the total value of all wagers [...].
        Games-played: accumulates
               ....
   </Requirement>
   <Requirement id="4"> Meters should be updated upon occurrence of
         any event that must be counted, including: play, cashout,
         bill in, coin in.
   </Requirement>
</Concern>
```

Composition rules are used to express crosscutting relationships. Listing 4.2 shows an example of composition rules, consisting of a `Constraint` tag that defines how the base requirements are constrained by aspectual requirements. The `Constraint` tag has *actions*, *operators* and *outcome* elements, used to express in detail how the base is affected. The action and operator tags informally describe how the base concern is constrained, imposing conditions in the composition. The operators express temporal intervals, temporal points or restrictions between sets of concerns. The outcome tags (*satisfied* and *fulfilled*) define the result of a composition. *Fulfilled* is used to denote that composition constraints have been imposed. *Satisfied* takes other requirement IDs as parameters and indicates that those requirements have been satisfied as a consequence of the imposed constraint [60].

Listing 4.2: A composition rule for Meters and Game using MDSOCRE.

```
<Composition>
  <Requirement concern="Meters" id="4">
    <Constraint action="enforce" operator="on">
           <Requirement concern="Game" id="3" />
    </Constraint>
    <Outcome action="fulfilled"/>
  </Requirement>
</Composition>
<Composition>
  <Requirement concern="Error Condition " id="7">
    <Constraint action="enforce" operator="on">
        <Requirement concern="Game" id="6" />
    </Constraint>
    <Outcome action="satisfied">
      <Requirement concern=" Error Condition" id="5"/>
       </Outcome>
  </Requirement>
 </Composition>
```

The first composition rule of Listing 4.2 shows how the *Meters* concern crosscuts the *Game* concern. In this example we have used the outcome action “fulfilled”, because there is no other set of requirements to be satisfied. This first composition rules means that when a play is awarded (according to the pay table) it must be counted by the corresponding meters. Other possible outcome action value is “satisfied” and the set of requirements that are satisfied. This is the case of the *Error Condition* composition, because when such a condition is detected an action must be taken, *i.e.,* an additional requirement has to be satisfied after the constraints have been applied. A concrete example of this is the second composition of Listing 4.2, which indicates that after a ticket is printed (cashout action indicated in R-SM-6) this event must be communicated to the on-line monitoring system (R-EC-5). This satisfies R-EC-7, which indicates that important events, including *ticket printed*, must be reported in real-time.

The granularity of the approach is adequate for our case study, since it is possible to clearly state which requirements are affected. The flexibility provided by the parameterized `constraint` tag helps to express different variants of crosscutting relationships even though MDSOCRE does not natively support this. We were able to combine actions and operators to document the interactions, as shown in Table 4.3.

We used the action *ensure* and the operator *with* to represent a *Dependency* interaction. This follows the informal definition by Moreira et. al. [60], that says that a certain condition for a requirement that is needed actually exists. We used the action *provide* and the operator *for* for *Reinforcement*, as it specifies additional features to a set of concern requirements. For *Mutex* the combination is the action *enforce*, to impose additional conditions with the operator *xor*, as we want to prevent the simultaneous activation of two implementations of the same functionality. Finally, note that for *Conflict* we used the same combination, since operationally the desired effect is to satisfy just one of the requirement sets involved (similar to *Mutex*). The downside of using this notation is that both interactions are documented in the same way, even though their semantics is clearly different.

4.3.6 Limitations of MDSOCRE

Although we were able to completely model the SM application, we were faced with two limitations of MDSOCRE which resulted in models that are sub-optimal. Firstly there is no explicit interaction relationship, and secondly there is no support for unifying disparate requirements. We discuss these limitations next.

Lack of Interaction Relationships

The actions and operators included in the composition rules only describe relationships between the crosscutting concern and the selected base concern. As we explained in Sect. 3.8, interactions occur even between concerns without a crosscutting relationship. In our case we need to express that *G2S* depends on the existence of *Meters* to report this information and also that having *Error Conditions* could reinforce the functionality of *G2S* enabling it to report a new set of events: error events. These interactions as well as mutex (see Sect. 3.8) are not explicitly supported by this approach.

As a workaround we have combined pairs of existing actions and operators, as shown in Table 4.3.

This solution however has three downsides:

1. It forces the use of composition rules even when no crosscutting is present, which seems contradictory with the original purpose of composition rules expressed by the authors: “they describe how a concern cuts across other concerns...” [60].

Table 4.3: Expressing Interaction types in MDSOCRE

	action	operator
Dependency	ensure	with
Reinforcement	provide	for
Mutex	enforce	xor
Conflict	enforce	xor

2. The expressiveness of our combinations is not optimal, as it is not easy to map the different interaction types with pairs of actions and operator. Consider, for instance, *provide for* compared to the word *"reinforce"*. *Reinforce* makes it explicit that the interaction is a positive influence to the other aspect, but we have to use *provide for*, which is only a way to try to represent this idea.

3. Using the semantics provided for actions and operators we could not find a way for explicitily expressing *conflicts*. This is probably due to the approach being focused on removing conflicts, hence no conflicts should need to be documented.

We consider the second downside to be a key factor. The mapping from Table 4.3 needs to be used to interpret implicit information that instead should be explicit. This makes such interpretation in this approach error prone, as we will show in Sect. 4.5.

Furthermore, the third downside has to do with inherent differences between the definition of *conflict* used by the approach, and the kind of conflicts found in the SM domain. As MDSOCRE includes a process for the elimination of conflicts, they cannot be clearly expressed using the existing actions and operators. Instead, our conflicts need to be documented and modeled. The form that conflicts in the SM domain is that of the *not and* (the negation of the logical *and*. That is a SM cannot have Meters and Demo at the same time. In contrast, mutex can be expressed as logical *xor*. For example, the SM can be configured from G2S or SCP, only one of these.

An alternative would be the use of meta concerns, which seem to be a natural place to store information regarding interactions. Extending MDSOCRE in this way implies adding information regarding the interaction type and the interacting concern. This could be done in a number of ways using XML, for example by adding a tag parameterized with two attributes: the interaction type and the name of the interacting concern. Meta concerns are not exactly aimed at this purpose, but with this small extension they can support the different kinds of interactions. The drawback here is a conceptual mismatch: meta concerns were designed to document generic concerns, but in our case, interactions are manifest in concrete concerns. We therefore did not use this alternative.

No Support for Unification

As in Theme, the ambiguity of requirements in the slot machine domain again impacts the process. Recall that in Section 4.3.3 we discussed the impact of having multiple and ambiguous requirement documents. For example, we may have different documents that list the meters or counters that must be provided by the SM, and furthermore in the same document meters can be defined in multiple requirements. In other words, the requirements of meters are scattered over multiple documents. It is however necessary to group all this information at design time to correctly design the meters concern.

One possibility here is to rewrite requirements so that meters listing is done just once, containing meter definition from all the sources. However, such a rewriting effort is only feasible when considering a small number of requirements. Furthermore, a downside of creating a unified list is that after a requirements fusion, it is hard to update these once (one of) the sources evolves.

Alternatively, we can keep requirements organized as in the originating document, removing the need for unification as well as the issue of requirements changes. However, it is desirable to have some kind of link between them, as these different requirements complement each other, *e.g.* they jointly form the list of all meters. This allows the engineer to analyze all the requirements defining the same concept. XML permits this, as it is possible to add cross-references between different elements in the tree. However, MDSOCRE lacks a facility for describing such cross-references.

4.4 Extensions of the Existing Approaches

4.4.1 Extensions to Theme/Doc: Theme/Doc-i

We now present some enhancements to Theme/Doc and give examples in the SM domain. These allow us to deal with the issues expressed in the previous section, as we will demonstrate next.

Granularity

In order to improve the information of requirements involved in a crosscutting relationship, we added *quantification labels* to the existing gray arrows of Theme/Doc. Quantification labels allow us to specify which requirements are involved in a given crosscutting relationship, from both sides: the crosscutting concern and the base concern.

As shown in Fig. 4.3, a quantification label has two parts separated by a colon:

Crosscutting requirements IDs: this a list, a range, or the keyword *all* that indicates which requirements are crosscutting in the concern where the arrow has its origin.

Base concern requirements IDs: this a list, a range, or the keyword *all* that indicates which requirements are the requirements affected by the crosscutting (the destination of the arrow).

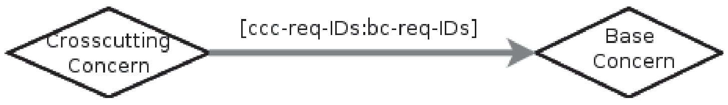


Figure 4.3: Quantification label applied to a crosscutting relationship

Before deciding in favor of quantification labels, we evaluated the use of graphical elements to indicate affected requirements. We considered, for example, adding arrows from affecting to affected requirements. We discarded such graphical approaches as we found that they add a large amount of clutter to the diagrams, making comprehension more difficult.

In summary, quantification labels allows us to express fine grained information that otherwise would be missed using the original Theme/Doc. This information can be used in later phases to take special care of the design and the implementation of the requirements involved in such relationships. For example, quantification labels used in the *base* side of a crosscutting relationship can help in the definition of pointcuts during design.

Expressing Interactions

As outlined in Section 4.3.3, we also need to model interactions between themes, since multiple themes need to be combined to form a system. As interactions are a new type of relationship between concerns, we decided on a notation that is consistent with the Theme/Doc notation that relates concerns. We propose to document interactions using dashed lines or dashed arrows, depending on whether the interaction is directional or not. To this line or arrow a text label is attached, indicating which type of interaction is represented (mutex, conflict, reinforcement or dependency).

The choice of a line or an arrow depends on the type of the interaction. For some interactions, such as *mutex* and *conflict*, there is no direction, since they are symmetric relationships, hence no arrow is needed. However, in the case of *reinforcement* and *dependency*, it is important to specify the direction of the relationship in order to understand which concern is reinforcing the other concern, or which concern depends on the other. Hence for those interactions arrows are used. The different kinds of interaction notations are shown in Fig. 4.4.

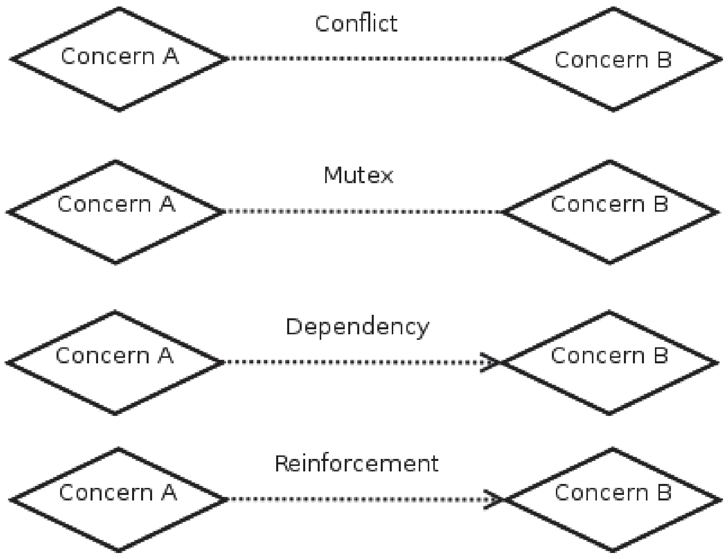


Figure 4.4: Interaction relationships

Interaction arrows can be combined with quantification labels in order to document which specific requirements are involved in the relationship. Examples of these combinations are shown in Sect. 4.4.1.

Implicit Requirements, Ambiguity of Requirements

Both issues of implicit requirements and the ambiguity of requirements benefit from the intervention of a domain expert. On the one hand, this will make explicit the requirements that are implicit, and on the other hand, this allows the requirements documents to be disambiguated. This implies adding some specific task in the process of Theme, probably during the requirements processing stage of Theme/Doc (where requirements are split, removed or added).

Considering the example of the implicit, derived requirement we presented in Section 4.3.3, we propose to add such requirements to the affected interaction itself. Fig. 4.5 shows how a derived time data source requirement is attached to the mutex interaction.

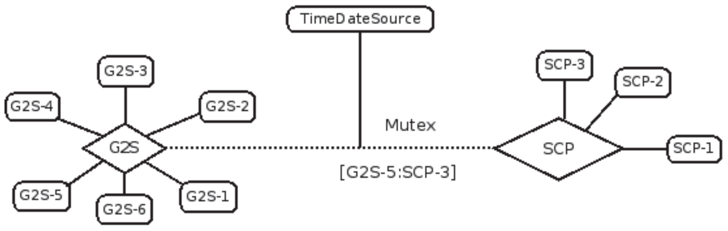


Figure 4.5: A new requirement related to a mutex interaction.

Theme/Doc-i Applied

Using our extensions, we were able to explicitly specify all the interactions in the case study. In this section we provide examples for the different types of interactions we have modeled. We refer to Appendix A for the full requirements model.

In Fig. 4.5 we have seen how a mutex interaction is represented. Fig. 4.6 shows how a conflict (explained in Sect. 3.8) is documented using the proposed extensions. Note that besides the interaction type, the requirements involved are specified using quantification labels.

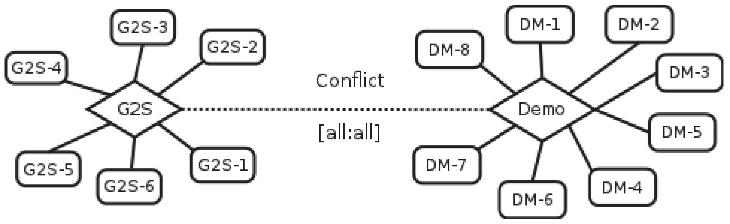


Figure 4.6: Conflict between Demo and G2S concerns.

Fig. 4.7 shows a reinforcement between *Error Conditions* and *G2S*. In this case, it is important to specify the direction of the relationship, since it should be clear which concern is reinforcing which other concern. Again, quantification labels have been used to identify requirements. From Fig. 4.7, it is clear that requirement 5 of *Error Conditions* enables extra behavior in the *G2S* concern, specifically it enables requirement 4 (*Devices can generate an event in unsolicited manner ...*).

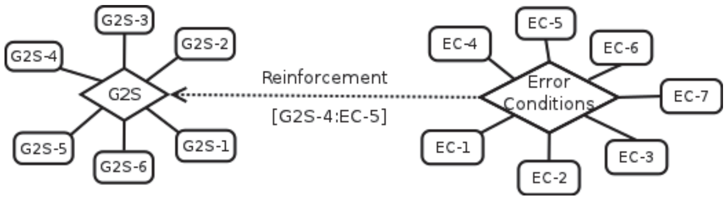


Figure 4.7: *G2S* concern reinforced by *Error Conditions* concern.

G2S (and any protocol that needs to report the SM state) depends on *Meters* to provide the required information. This situation is documented in Fig. 4.8, which shows that in order to satisfy requirement 2 of *G2S* first requirements 4 and 5 from the *Meters* concern need to be satisfied.

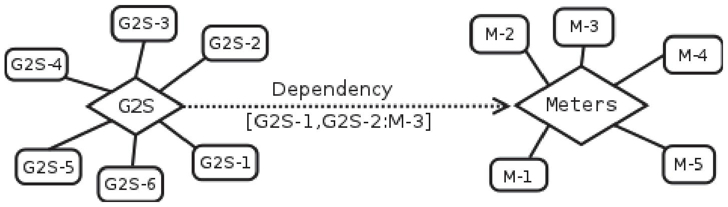


Figure 4.8: *G2S* concern depends on *Meters* concern.

4.4.2 Extensions to MDSOCRE: MDSOCRE-i

Considering the shortcomings of MDSOCRE outlined above, we propose an extension to MDSOCRE, called MDSOCRE-i. It consists of two parts: explicit interaction relations to address the need for clear notation of interactions, and cross-references to allow unification of scattered requirements into one complete set of requirements for a concept.

Explicit Interaction Relations

Unfortunately, MDSOCRE does not natively support the notion of interactions. Although they can be expressed using a combination of actions and operators, as in Table 4.3, we do not consider this a clean solution. This is because interactions are not documented in a way that makes it easy to differentiate between the different types of interactions (as confirmed by experimental results in Sect. 4.5). Note that in our requirements engineering we were forced to disregard parts of the MDSOCRE methodology: originally conflicts are not supposed to be expressed in the resulting models as they should have been removed as part of the requirements engineering process. Instead we need these conflicts to be explicitly present.

An alternative to our workaround is the use of the meta concern facility to store information regarding interactions. However there is a conceptual mismatch here as it means storing concrete information in an artifact that is aimed at expressing generic information regarding concerns.

We assert that a new relationship between concerns, aimed at documenting interactions, is needed. This new relationship would enable us to express interactions between concrete concerns, as well as between meta concerns when necessary. We propose an extension for MDSOCRE to effectively document interactions and call this extension MDSOCRE-i.

Before deciding for the extension presented here we evaluated different solutions. For example, having new actions or operators was considered as a possible solution. We however discarded this because some combinations of existing actions and operators with new actions or operators would not make any sense. Another possibility we evaluated was to replace the `Composition` XML tag by an `Interaction` tag, which would be a much more intrusive change to the MDSOCRE specification. Moreover, interactions form part of the information of a composition, and therefore this option was also discarded.

We propose to extend the `Composition` XML element of MDSOCRE, allowing it not only to express crosscutting relationships but also interactions. In listing 4.3, we show a new `Interaction` XML element that can be parameterized with the interaction type. We do not pose any restriction on this type, and here simply specify the interaction types as used in the previous sections of the text. Note that we only apply this extension at the base level, but it can also be extended to apply at the meta-concern level.

The interaction element is always contained in a requirement element and nests a second requirement element. The containing requirement element affects the nested requirement element as specified by the type of the interaction element. In other words, the direction of the interaction relationship (when relevant) is from the outer requirement to the inner one. Consider for example Listing 4.3 that describes the dependency between G2S (requirement 6) and Game Recall (requirements 1 and 2). As we explained in Sect. 3.8, *G2S* depends on *Game Recall*, as it needs the information captured by *Game Recall* in order to communicate it to the on-line monitoring systems when needed.

Listing 4.3: The `Interaction` element instantiated for a *dependency*

```
<Composition>
 <Requirement concern="G2S" id="6">
  <Interaction type="dependency">
   <Requirement concern="GameRecall" id="1,2"/>
  </Interaction>
 </Requirement>
</Composition>
```

Note that the `Requirement` tag has not been modified with respect to the original MDSOCRE specification as it already allows specification of which requirements participate in a composition.

Listing 4.4 shows examples for the other three interactions (reinforcement, mutex and conflict).

Listing 4.4: Reinforcement, conflict and mutex interactions

```
<Composition>
 <Requirement concern="Meters" id="2,3">
   <Interaction type="reinforcement" >
     <Requirement concern="Proprietary Communication Protocol" id
         ="4"/>
  </Interaction>
 </Requirement>
</Composition>

<Composition>
 <Requirement concern="Demo Mode" id="1,4">
    <Interaction type="conflicts" >
      <Requirement concern="Game Recall" id="1,2"/>
    </Interaction>
 </Requirement>
</Composition>

<Composition>
 <Requirement concern="Demo Mode" id="9">
    <Interaction type="mutualExclusion" >
      <Requirement concern="Game" id="3"/>
    </Interaction>
 </Requirement>
</Composition>
```

Note that in the case of mutex and conflict interactions the relationships are symmetrical, so the order of the requirements may vary with no impact: It is equivalent to say that Game Recall conflicts with Demo or Demo conflicts with Game Recall. In the case of mutex, the symmetrical nature of the relationship is more obvious as it is used to document two (or more) instances of a given functionality. For example, Listing 4.4 documents the mutual exclusion between two requirements, one of the Game concern and one of Demo concern. The mutual exclusion is required because both of them determine a different way of generating the outcome for a play.

Cross-references for Unification

As indicated in Sect. 4.3.6, in the SM domain requirements coming from different sources describe the same concepts. These different sources need to be unified to eliminate ambiguities and to ensure that the requirements model is complete. This is because requirements may complement each other while originating from different documents, or from different parts of the same document. Furthermore changes in a requirement may impact all the requirements that it complements, in the same document as well as in other documents. To allow such unification and a more straightforward assessment of the impact of changing a requirement, MDSOCRE-i contains a second extension: adding cross-references to a requirement.

The cross-reference extension is meant to be used inside of a single concern to link requirements describing different perspectives of the same concept. Our extension consists of a `seeAlso` attribute that lists the identifiers of related requirements, *i.e.,* specifying a cross-reference. When adding complementary requirements to an existing set of requirements, the requirement engineer adds this attribute to the requirement tags. This attribute is added both in the existing set of requirements, referring to the newly added requirement that complements them, as well as to the new requirement, referencing the old requirements that are being complemented.

In Listing 4.5, we show an example of how a requirement defining some meters, taken from the G2S documentation, refers (using the `seeAlso` attribute) to requirements 1, 3 and 4 of the same concern, defined in the GLI documentation that contain complementary information.

Listing 4.5: See also extension

```
<Concern name="Meters">
<!-- From GLI 11 -->
 <Requirement id="1" seeAlso="3,4,5"> Credit meter: shall at all
     times indicate all credits or cash available for the player to
     wager or cashout.
 </Requirement>
 <Requirement id="2"> Credit Meter Incrementing: The value of every
     prize (at the end of a game) shall be added to the player's
     credit meter. The credit meter shall also increment with the
     value of all valid coins, tokens, bills, Ticket/Vouchers,
     coupons or other approved notes accepted.
 </Requirement>
 <Requirement id="3" seeAlso="1,4,5"> Accounting Meters:  Coin In: a
      meter that  accumulates the total value of all wagers [...].
     Games-played: accumulates  the number of games played; since
     power reset, since door close and since game initialization.
 </Requirement>
 <Requirement id="4" seeAlso="1,3,5"> Meters should be updated upon
     occurrence of any event that must be counted, including: play,
     cashout, bill in, coin in.
 </Requirement>
<!-- From G2S Docs-->
 <Requirement id="5" seeAlso="1,3,4"> Some G2S meters are:
     gamesSinceInitCn Number of games since initialization. WonCnt:
     Number of primary games won by the player. LostCnt: Number of
     primary games lost by the player.
 </Requirement>
</Concern>
```

Applying MDSOCRE-i

We successfully used MDSOCRE-i to build a complete requirements model of the SM requirement subset presented in Sect. 4.2. We were able to model all interactions present in the requirements, and to use cross-references to build complete sets of requirements for all concepts that are defined in a scattered form. Considering the size of the specification documents, we found that MDSOCRE-i yielded slightly shorter documents: 228 lines versus 236 lines for MDSOCRE, even though extra cross-reference information is available. The full model is included in Appendix B.

4.4.3 Summary of Extensions

In this section we have proposed a number of extensions to both Theme/Doc and MDSOCRE to address the weaknesses we encountered when performing

Table 4.4: Limitation of Theme/Doc and proposed extensions.

Limitation in Theme/Doc - Sect. 4.3.3	**Extension in Theme/Doc-i** - Sect. 4.4.1
Granularity	Quantification labels
Expressing Interactions	New arrow notation combined with quantification labels
Ambiguous and implicit requirements	Disambiguation, derived requirements

requirements analysis. The tables 4.4 and 4.5 summarize the weaknesses of the approaches we studied and the extensions we made to address them.

Table 4.5: Limitation of MDSOCRE and proposed extensions.

Limitation in MDSOCRE - Sect. 4.3.6	**Extension in MDSOCRE-i** - Sect. 4.4.2
Lack of explicit interaction relationships	New tags and attributes to describe interacting concerns and requirements
No support for unification	New tag and attribute to document relationships between requirements describing closely related topics.

The extensions we propose to Theme/Doc allow us to express information that otherwise would not be present in the requirements specifications we produce. Our extensions to MDSOCRE that yield MDSOCRE-i however do not add new expressive capabilities for interactions to the methodology. Instead, they aim at helping the requirements engineer to easily understand interactions as contained in the requirements specifications we produce. In order to validate this hypothesis, we performed a user study that is presented next.

4.5 User Study: MDSOCRE, MDSOCRE* and MDSOCRE-i

As we have discussed above, both MDSOCRE and MDSOCRE-i are sufficient to be able to model interactions, thanks to the use of the mappings specified in Table 4.3. We call the variant of MDSOCRE that includes these mappings MDSOCRE*. The aim of MDSOCRE-i however goes beyond MDSOCRE*: it also aims to achieve a conceptually cleaner fashion to model interactions, leading to a modeling phase that is faster and a model that is more complete and less error-prone to interpret. We have performed an additional comparison between MDSOCRE (including MDSOCRE*) and MDSOCRE-i to verify whether this is indeed the case, in the form of a restricted user study.

In our user study we compared interaction support of the two approaches in terms of accuracy and speed by means of two experiments. In other words, we compared legibility of the explicitly labeled interactions versus the action-operator pairs we used in Section 4.3.6. In the first experiment we established whether or not the action-operator pair notation has legibility issues, by evaluating the advantages of using Table 4.3 (*i.e.,* using MDSOCRE*) in the requirements interpretation process. We found that the use of Table 4.3 significantly improves legibility of the requirements model. In the second experiment, we therefore compared MDSOCRE-i against MDSOCRE*, to see if MDSOCRE-i outperforms this setup. We found that indeed, even in this case, MDSOCRE-i is a significant improvement.

In each of the two experiments we tested legibility of the four interaction types as follows: In the first phase we established legibility of the first notation, and in the second phase of the second notation. Each phase consisted of four tasks, covering the four interaction types. In each task two concerns were presented with their interaction described in the corresponding notation. Then, a multiple choice of three different interactions between the two concerns was presented. The subject had to select which of the options was correct, yielding a measure of accuracy. The time needed to solve each task was also measured, in order to establish which approach delivers faster results while working with interactions. For both experiments all the subjects were in the same room. A single clock was available so that the subjects could take note of the current time for each individually finished task. The final times taken for each task were calculated by taking the difference between the time of the current and the previous task. Lastly, in order to evaluate the subjective preference for each approach, after executing each experiment, a survey was completed by the subjects. It inquired as to their preference of notation per interaction type, graded on a five-point Likert scale.

The details of each case study are described in the following sections.

4.5.1 Case Study 1: MDSOCRE vs. MDSOCRE*

The first experiment we performed aimed to evaluate the original MDSOCRE, without knowledge of the semantics of the operator-action pair notation given in Table 4.3, versus MDSOCRE where this semantics is explicitly defined, which we call MDSOCRE*. Note that MDSOCRE* does not make any modifications to the MDSOCRE notation, the only difference is the semantics which is explicitly defined in Table 4.3.

Evaluators The subjects of the study were seven experienced IT professionals, but not knowledgeable in the SM domain. Each of them at least worked for four years using requirements, most of them in the role of requirements analyst. The group has an average experience of 11.5 years working on commercial IT projects.

Activities The requirement interpretation tasks presented to the subjects were not taken from the SM domain, to avoid confusion due to unknown terminology. Instead, generic requirements were generated for their interpretation. For example, Listing 4.6 shows the requirement specification testing *Dependency*, in the second phase of the experiment treating MDSOCRE*. The multiple-choice options given to the subject for this task were the following (the correct answer is the third option):

1. User history features can be used without the availability of storage facilities.
2. User history benefits from the availability of storage facilities when available.
3. User history features depend on the availability of storage facilities.

Listing 4.6: Task evaluating Dependency in MDSOCRE*

```
<Concern name="Context-awareness">
 <Requirement id="1">Context-monitoring is the repeated observation
     of an entity’s context through an input mechanism.
 </Requirement>
<Requirement id="2"> User history is tracked through an input
    mechanism, and it has to be retrieved afterward.
 </Requirement>
</Concern>

<Concern name="Persistence">
 <Requirement id="1">State-encoding is the conversion of application
      data to a format more suitable for storage in a database
     management system.
 </Requirement>
</Concern>

<Composition>
 <Requirement concern="Persistence" id="1">
  <Constraint action="ensure" operator="with">
   <Requirement concern="Context-awareness" id="2"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
```

The overall organization of the experiment was as follows:

1. The four types of interactions were presented.
2. The MDSOCRE approach was explained, including usage examples.
3. The first set of tasks was given to the subjects for resolution.
4. After all the users finished solving the first set, the mappings for interactions, *i.e.,* MDSOCRE*, was explained.
5. A second set of tasks was given for resolution.

Survey After all tasks were completed, study subjects were presented a questionnaire. It asked, for each interaction type, whether they would use a specific notation in the future to document this kind of interaction, graded on a five point Likert scale. Sentences followed the pattern below:

> I would use **MDSOCRE** to express *dependency* — *reinforcement* — *mutex* — *conflict* interactions in the future.
> I would use **MDSOCRE*** to express *dependency* — *reinforcement* — *mutex* — *conflict* interactions in the future.

Table 4.6: Results of case study 1: MDSOCRE (M) versus MDSOCRE* (M-*), and statistical analysis of population independence of subjective evaluation.

Interaction	Correctness		Time		Subjective Evaluation		
	M	M-*	M	M-*	Use M	Use M-*	p-value
Dependency	85.71%	71.42%	3m 43s	2m 34s	2.28	4.14	0.006
Reinforcement	57.14%	71.42%	3m	3m 09s	2.2	4	0.011
Conflict	57.14%	85.71%	3m 51s	3m	1.8	4.14	0.002
Mutex	71.42%	100%	2m 26s	1m 43s	2	4.28	0.002
Global	64.28%	85.71%	13m	10m 26s	2.10	4.14	

Results The results of the experiment regarding accuracy, time and subjective preference are presented in Table 4.6. Global results show that MDSOCRE* is amply more accurate (85.71% against 64.28%) for the whole experiment. Broken down by interaction type, MDSOCRE* is more accurate for *Reinforcement*, *Conflict* and *Mutex* interactions but slightly less accurate for *Dependency*. Regarding the time taken, MDSOCRE* provided significantly faster results, except for *Reinforcement*, where the time taken was slightly more than MDSOCRE. Overall time taken shows that MDSOCRE* is 24% faster than MDSOCRE. In the subjective preference score we see a strong bias towards MDSOCRE*, both per interaction type as globally. Moreover, statistical analysis reveals that the difference between the opinions for both notations is highly significant. We performed a population independence test[3] to establish this for each interaction type. The *p-values* for the tests show that with a 98% confidence level we can say that the opinion of the test subjects for MDSOCRE is different than for MDSOCRE*. Lastly, overall results show that the study subjects would use MDSOCRE* in the future, while they did not agree to use MDSOCRE in the future.

To conclude, the experiment shows that interpreting interactions in a requirement model written in MDSOCRE* is not only preferred by the users, but also significantly faster and also more accurate than when written in MDSOCRE.

4.5.2 Case Study 2: MDSOCRE* vs MDSOCRE-i

The second experiment we performed evaluated the advantages of our extension to unmodified MDSOCRE. As the previous experiment showed that MDSOCRE* performs better than MDSOCRE, we chose to compare MDSOCRE* with MDSOCRE-i.

Evaluators To avoid that learning effects from the first experiment influence the results of this experiment, a different group of test subjects was used. The group for this experiment consisted of eight people with industrial experience in the SM domain. All of them have worked for a company developing and testing SM software during an average of 4 years. They occupied different positions, *e.g.* testers, architects, developers. For this experiment we therefore used concerns and interactions taken from our SM requirement models written in MDSOCRE and MDSOCRE-i (included in Appendix B).

[3]A two independent sample Wilcoxon rank sum test.

Table 4.7: Results of case study 2: MDSOCRE* (M-*) versus MDSOCRE-i (M-i), and analysis of population independence of subjective evaluation.

Interaction	Correctness		Time		Subjective Evaluation		
	M-*	M-i	M-*	M-i	Use M*	Use M-i	p-value
Dependency	75%	87.5%	4m	4m 24s	2.5	3.5	0.36
Reinforcement	75%	100%	5m 28s	2m 23s	2.25	3.5	0.46
Conflict	87.5%	100%	2m 22s	2m 35s	2.5	3.87	0.007
Mutex	87.5%	87.5%	1m 54s	1m 59s	2.25	3.87	0.005
Global	81.25%	93.75%	13m 44s	10m 21s	2.37	3.68	

Activities The experiment was organized as follows:

1. The four types of interactions were presented.
2. MDSOCRE approach was explained, including usage examples.
3. The mappings of MDSOCRE* for interactions were introduced.
4. MDSOCRE-i was introduced, including usage examples.
5. The two sets of tasks were delivered for their resolution.

Survey After the tasks were completed, a similar survey as in the first experiment was handed to the subjects (substituting MDSOCRE-i for MDSOCRE in the questions). This survey also included the following questions, allowing a response of either MDSOCRE* or MDSOCRE-i:

Which notation makes interactions more evident?
Which notation describes the interactions more clearly?

Results The results for the second case study are presented in Table 4.7. Correctness is shown in the two first columns. Globally, MDSOCRE-i results in a higher accuracy than MDSOCRE*. Considering the interaction type, for *Dependency*, *Reinforcement* and *Conflict* MDSOCRE-i performed better while for *Mutex* the correctness results are the same. With respect to time, MDSOCRE-i was notably faster for *Reinforcement*, while for the other interactions the results are equivalent. In the overall time measure MDSOCRE-i was 32% faster than MDSOCRE*. Regarding subjective preference, users mildly agree to use MDSOCRE-i in the future while mildly disagreeing to use MDSOCRE* in the future. With less pronounced differences between the opinions of both notations, statistical analysis reveals that the difference between the opinions for both notations still is significant. The *p-values* for the tests show that with a 95% confidence level we can say that the opinion of the test subjects for MDSOCRE* is different than for MDSOCRE-i.

Finally, for the last two questions of the questionnaire: which tool makes interaction more evident and which tool describes interactions more clearly, on both questions MDSOCRE-i got 87.5% of preference compared to 12.5% obtained by MDSOCRE.

From the second experiment, we can conclude that MDSOCRE-i is preferred by the users, while being somewhat more accurate and significantly faster than MDSOCRE*.

Overall, considering the time and accuracy measurements of both experiments we can therefore conclude that for the interpretation of interactions in a requirements model MDSOCRE-i is significantly more accurate than MDSOCRE while also being much faster. Finally, from the subjective evaluation of MDSOCRE* being better than MDSOCRE, and of MDSOCRE-i being better than MDSOCRE*, we can infer that users largely prefer MDSOCRE-i over unmodified MDSOCRE (which includes MDSOCRE*).

Time Taken Using our extensions implies that more time needs to be invested in the explanation on how to use these new constructs. Even though we did not accurately measure the time needed for this *teaching* part of the experiment, we do have some observations in this regard. After the engineers were exposed to the limitations of the original approach and our workaround, they welcomed our extensions and quickly understood how to use them. Roughly speaking, the explanation of the extensions took one third of the time needed for the original introduction of the approach (which includes the workaround). Lastly, compared to the time to explain the extensions, the explanation of the workaround took more time, in addition to being more cumbersome and more error prone.

Study Validity: Strong and Weak Points

Strong Point: Industrial Profile Because the MDSOCRE-i extension has been inspired by industrial experience in a complex domain, having the proposed extension validated by people in industry is fundamental. The strongest point of our study is therefore the industrial profile of the subjects.

In both cases the subjects had considerable experience in the industry. In the case of the first experiment, the test subjects belong to a company which develops mainly for the enterprise domain. They were accustomed coping with interactions of requirements, frequently present in enterprise applications.

For the second case study, the subjects were experts for the particular domain of SMs. Moreover, they occupied different positions in the company, which reinforces the hypothesis about the ability of MDSOCRE-i to improve comprehension, not only for requirement engineers, but also for testers, developers and other profiles.

Weak Point: Small Scale The main weakness of both our studies is that they were performed with a small group of test subjects (seven and eight respectively). Given the industrial setting, we were unable to locate a larger group of people for both tests. Moreover, for the second test, the use of domain experts further restricted the set of possible test subjects.

However we do consider the study to be relevant because we see that the results are very consistent. Our assessment is confirmed at least regarding the user preference since the statistical analysis of the subjective evaluation returns highly significant results. In other words, we can conclude that with a bigger sample size it is extremely likely that the user preferences would also be in favor of MDSOCRE-i.

In conclusion we state that despite the small number of subjects the results regarding time, accuracy and subjective evaluation demonstrate that MDSOCRE-i significantly outperforms MDSOCRE with respect to the comprehension of interactions in requirement models.

4.6 Conclusions

This chapter presents our study of applicability of two AORE approaches: Theme/Doc [8] and MDSOCRE [60], in the Slots Machines Domain. We focused mainly on the expressiveness of these approaches in terms of interactions between requirements. We found that both approaches lacked comprehensive support for our case, and proposed and validated extensions to both approaches to address this issue.

From our analysis we conclude that, considering both approaches without our extensions, MDSOCRE performs better than Theme/Doc. This because it allows specification of the composition of concerns in detail. We also noticed a considerable difference in the process for attaching requirements to their concerns. Theme/Doc relies on the analysis of the text of requirements, searching for the concern name, while MDSOCRE relies on the analyst's domain knowledge. As we have different sources with different terminology, we found the MDSOCRE approach more suitable for our needs.

The first limitation of Theme/Doc is the lack of detailed information regarding the requirements participating in crosscutting relationships. To address this we added quantification labels that allow the analyst to provide more detail to concern relationships. A second issue is the lack of support for expressing interactions between concerns. In order to solve this we introduced a new kind of interaction relationship that permits to express a conflict, mutex, dependency or reinforcement between two concerns, information that otherwise would be lost. This new interaction relationship can be combined with quantification labels to accurately document the requirements involved in the interaction.

For MDSOCRE, we were able to model interactions between two concerns using specific combinations of action and operator attributes in constraint specifications. As these combinations can be ambiguous and seemed unintuitive, we extended MDSOCRE with explicit support for interactions, called MDSOCRE-i. We then performed a user study to establish this ambiguity and to validate that MDSOCRE-i delivers faster results and aids in the understanding of interactions.

Having identified and documented the interacting concerns we expect this information will aid in the development of the architectural and design artifacts that will allow the produced application to correctly address interactions between these concerns. We next present our work on the design phase in the Slot Machines domain.

Chapter 5

Interactions in Design

This chapter is based on our previously published work *Expressing Aspectual Interactions in Design: Experiences in the Slot Machine Domain* [36].

In the previous chapter we studied interactions in the context of requirements analysis. The next step is modeling the software using an adequate approach for *Aspect Oriented Modeling* (AOM). However, to the best of our knowledge there has been no work published that evaluates AOM approaches in an industrial setting with a focus on interactions between the different concerns. We therefore undertook an evaluation of two mature AOM approaches to establish their applicability in our context. Somewhat surprisingly, neither of these two is adequate in our setting, as we report in this chapter.

As basis for our selection we used surveys on AOM [20, 75], complemented by a study of more recent literature. The chosen approaches are Theme/UML [26] and WEAVR [28, 29]. Beyond their maturity, acceptance in the AOM community, and claimed support for interactions, both methodologies have specific advantages. Theme/UML integrates with Theme/Doc: an aspect-oriented methodology for requirements specification. WEAVR is arguably the best-known industrial application of AOM [94], and the only methodology that we are aware of that is used in industry to develop complex applications.

Other approaches has not been included due to the lack of interaction modeling support. MATA [93] provides interaction – particularly conflicts – detection capabilities based on a model of influences. After detection, conflicts must be removed. The JAC Design Notation [66] has been tested against industrial cases, but it does not support interactions. UML based approaches, such as Aspect-Oriented Software Development with Use Cases [48], recognize the existence of conflicts, but they claim that conflicts must be avoided, or planned its support as future work [76].

This chapter is organized as follows: Sect. 5.1 presents our needs considering design documents in the context of our problem. Next, Sect. 5.2 presents some design decisions for the SM. Section 5.3 then proceeds with an evaluation of Theme/UML, and Sect. 5.4 follows up with an evaluation of WEAVR. We present our conclusions and future work in Sect. 5.5.

5.1 Requirements for the Design

5.1.1 What is Expected from the Design Document

In the design phase, our goal is to refine the requirement specification documents into a model of the software artifacts that will form the final system. This model, written down in a design document, will be passed to the developers for implementation. Hence, it should be sufficiently complete to allow for the implementation to be produced relatively independently. As we are performing Aspect-Oriented Software Development, the choice of an AOM approach for creating this document is a given. We expect that we will be able to produce the complete design documents, *i.e.,* not having to resort to a significant amount of additional documents with an ad-hoc notation to complement for omissions in the methodology. In the latter case, the advantages of using a standard AOM are small and we would consider rolling our own AOM. We furthermore have two related expectations of the design document: maintenance support and explicit interactions.

In subsequent maintenance or evolution phases, the changes made in the requirements will trigger subsequent changes in the design, and the developers will modify the implementation accordingly. Such later modifications may not break the system because they violate constraints of the original design or go against the original design decisions. If the change is significant enough to warrant modifying the design constraints or assumptions, the original intentions should be maintained as much as possible. Hence the design document must be clear on which are the the critical design decisions that were made and what assumptions were taken. Furthermore, it is known that the presence of aspects in a software system that is evolving can be problematic [51]. Such issues should be mitigated by the information that is explicitly available in the design document. When evolving the software the implementers must be able to use the document as a guide, seeing what assumptions taken by the aspects no longer hold, or what new code now also falls within the realm of an aspect.

As we have said above, our experience is that there is a significant amount of non-trivial interactions between the different aspects of the system. This is also confirmed by the results of the requirements analysis we have reported in Chapter 4. Even though aspects are intended to provide advanced modularity and decoupling, they do not exist in isolation. As any module in software, their presence impacts other modules and their functionality may depend on other modules. Documented design decisions should therefore include not only which modules will be aspects and where they crosscut, but also how they *interact* with each other. This information must be made explicit so that critical information is correctly passed to the implementation phase, and is present when maintaining or evolving the software.

5.1.2 Scalability is Key

Recall that in Sect. 4.2 we mentioned that the SM application requirements documents establish approximately 600 requirements [99]. This results in a crucial need for scalability of the design phase. We consider it unrealistic to produce a design document that goes into great detail for all of these requirements, as such a heavyweight approach will not scale.

A second motivation for the need for scalability is that the different requirement specifications [40, 41, 63] regularly change and moreover, are under control of different legal institutions. As a result, changes to deal with new (legal) issues are not synchronized between the different documents. A heavyweight design document that needs to be updated on each change of a regulation document as well as consequent changes in other documents will cause an unacceptable overhead in maintenance and evolution.

As a partial solution to handle these scalability issues, we expect the AOM approach to provide for some means of abstraction over similar patterns in the design. For example, there are different (informal) types of errors that can occur in the SM, and each type requires a different action to be undertaken. The two extreme cases of errors are the following: Minor errors, such as the ticket printer running out of paper, requires a message to be sent to the casino server without interrupting play. Major errors, such as a player tilting the machine (to attempt to influence the outcome of a play) requires the machine to lock up immediately, and call an attendant by lighting the lamp at the top of the machine while sounding an alarm. There should be a way such that for a class of error only one model is created, instead of a model for each specific error condition.

5.2 Design Overview

Considering the results of the requirements analysis phase we previously performed, we now give an outline of how we envision the design of the SM software. This provides us with a concrete basis for evaluation of the AOM, as it must allow us to expand and refine this overview into a complete design document.

5.2.1 Aspects in the Design

A class diagram that shows the outline of the design is given in Fig. 5.1. It uses an ad-hoc extension of UML to indicate crosscutting, showing that we model the following crosscutting concerns as aspects: Metering, Demo, Program Resumption, Error Conditions, SCP Protocol, G2S Protocol. For a description of these concerns, see Sect. 3.7 The following is an overview of their crosscutting nature:

Metering The Metering aspect crosscuts Game and other base entities in order to keep meter data up to date.

Demo The SM must have a "Demo" mode, where all possible outcomes for a play can be simulated. The Demo concern needs to control the outcome produced by the Game class or the Random Number Generator. It furthermore crosscuts Metering to avoid polluting accounting meters when it is active. It also needs to avoid communication protocols (G2S and SCP) reporting data during a demo run (as it would report fake events).

Program Resumption Information to be saved includes the status of the current play and the values of the meters and events which are pending for

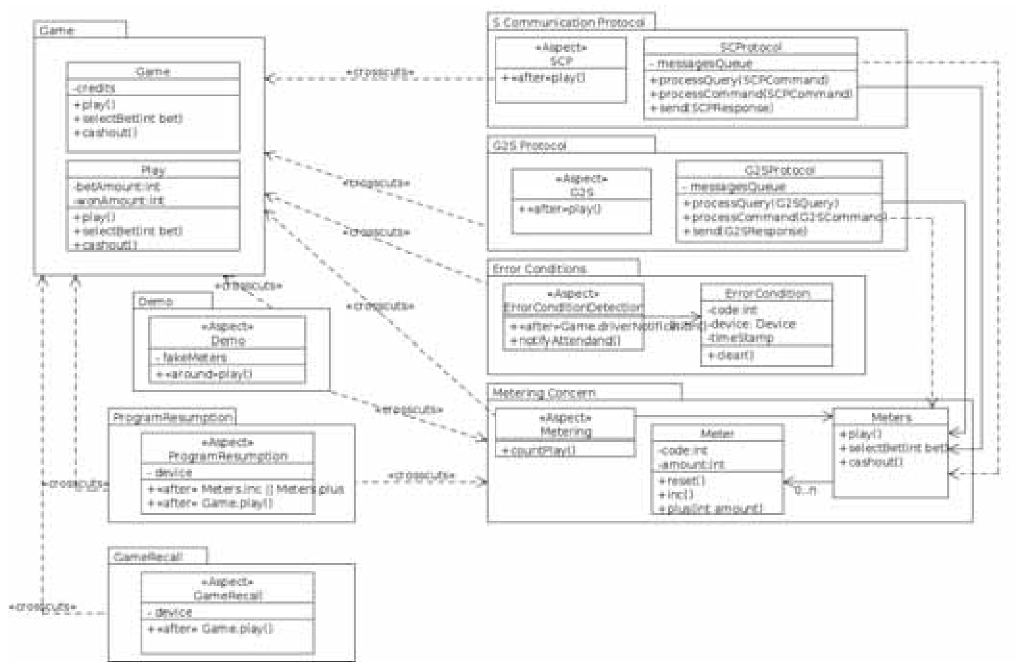


Figure 5.1: Overview of the class structure of the design.

reporting to the monitoring system. The system should recover the last state or setting after a power outage.

Error Conditions Error conditions detected by the game such as: tilt, out of paper, ... are detected by the Error Condition Detection aspect. Once an error condition is detected some actions need to be performed, *e.g.* in case of a tilt illuminating the tower lamp and sounding an alarm to call the casino attendant.

GameRecall Information regarding credits, reels position and prize awarded is saved for each play.

Communication Protocols These aspects crosscut the Game modules to add behavior such as multiple SMs vying for the same jackpot. Moreover, both protocols need to report metering information and hence crosscut the Meters aspect.

Figure 5.1 shows that a simple extension of UML already suffices to provide the outlines of the aspectual design. Not surprisingly, most, if not all, of the AOM approaches we studied, allow us to produce a model similar to this diagram. What is however lacking in the above diagram is the information of how the various aspects interact with each other, as well as with the base application. For example, when in Demo mode network communication must be disabled, as queries from the server may only receive values corresponding to normal play conditions. This information should also be present in the design document, but we find no immediately obvious way in which this can be diagrammed, hence the lack of this information in the Figure 5.1.

5.2.2 Interactions Between Concerns

Our resulting design document not only needs to contain the information of the aspects present in the system and how they crosscut, but also the interactions between concerns. That is, It is also necessary that the interactions which were identified in the requirements analysis phase be present in the design document. Furthermore, the strategies envisioned to treat those interactions need to be expressed.

To better understand what our needs are for this part of the design document, now we briefly recap on the different interactions in the SM, and how we want this to be reflected in the document.

At a lower level of abstraction, we need to specify how the crosscutting behavior and the interactions must be operationalized. This will depend on the final implementation platform. Therefore we prefer not to go into too much detail at this point. Instead, we present a general design on how the interactions should be resolved.

Conflict between Demo and Multiple Concerns

As explained in Sect. 2.2 a conflict arises when aspects add incompatible behavior. In our case it is not possible to completely solve the conflict by removing part of the behavior, as both are required. Instead, we need to implement the system in a way that avoid the incompatible behaviors to be present a the same time. This is what we call a conflict resolution strategy.

The Meters, Communication Protocols and Program Resumption aspects are present to comply with legal accounting requirements regarding plays performed on a SM. The Demo aspect, also a legal requirement, conflicts with all of the above mentioned aspects. The conflict originates from the regulation, which states that a play in Demo mode must not alter the meters nor that its activity is visible over the network to the monitoring system, nor saved into persistent storage. Hence, after a Demo session, the Game must recover its original state and any event or state change while in Demo must not be reported by the communication protocols to monitoring system.

The key behavior for our Demo aspect is to influence how the Game and the outcome generator decide the prize to be awarded. At the same time, from the conflict standpoint, it interacts with other aspects to keep system data consistent according to regulations, avoiding the pollution of critical information with the information generated during the Demo session.

Design decisions In order to provide the Demo behavior and treat its conflicting behavior, design decisions may include:

1. To implement the core behavior of the Demo mode, around advices will be used to capture and alter the behavior of methods that determine the generated outcome for a given play.

2. Communication Protocols must report the SM as being in *out of service* state, so that no "demo" information is reported to the backends and monitoring systems. This might be done by intercepting the responding

behavior of the protocols and responding with an "out of service" message[1].

3. Meters and GameRecall information must not be polluted with Demo information. A possible solution is to keep objects representing the state in a safe place, and replace them by fake ones, where data generated during Demo will be stored. On the finishing of the Demo session, the "original objects" must be restored.

4. Program Resumption must not persist the information generated during "demo" plays. The functionality of this aspect must be deactivated. That is, if the machine is re-started in normal mode, changes due to demo mode must be lost.

Mutex for Communication Protocols

Both communication protocols provide similar functionality: allowing the monitoring system to *query* information and *set* some configuration values and state on the SM. For query commands, such as reporting the value of certain meter or requesting information regarding the last plays, there is no problem with having both protocols active at the same time, as they will not interfere with each other. Conversely, for operations that modify the state of the SM, mutual exclusion must be ensured during a single program execution. If not, inconsistencies in the SM internal state may arise. For example, consider setting the time of the SM, an operation performed by the casino server. With both communication protocols enabled, two different servers with different clock values may set the time on the SM to either of both clock values. As a result, the time-stamps of events on the SM will be inconsistent because newer events may show a previous time-stamp to older ones. The same happens with a progressive prize accumulator, which is sent to the SM by the monitoring system. This value is presented in some displays of the SM. If the SM processes the progressive value coming from two different servers (which for sure will have different progressive values), it could result in the SM showing (and possibly paying!) an incorrect progressive prize.

To document this *mutex* it is necessary to state that certain execution traces must not occur during a single SM run session.

Design Decisions

- Different server commands will be objects of different classes, which will belong to the same hierarchy as needed. For example: SetProgressive, SetTime, etc.

- As protocols use almost the same set of commands, these will be shared between the protocols. That is, there will be no separate hierarchies for G2S and SCP commands. This means that a command such as SetTime can be issued either by G2S or SCP protocol.

[1]All the communication protocols (there are others apart from those treated in this work) allow to report the machine as being out of service. Stopping communications completely is not an option due to protocol constraints.

- For a given run of the system, it is necessary to assign which command can be received from which protocol. This can be done by configuration, or using an alternative policy, for example: once a command (let's say SetTime) is received through a protocol (for instance G2S), the next occurrences of SetTime will only be processed if they come from the G2S monitoring system.
- Complementary, if a configuration command arrives through an improper protocol (SetTime coming from SCP in our example), it will be ignored, and this occurrence will be logged for future fixing.
- Changes in the configuration can be assumed to occur when the SM is restarted.

Reinforcement of Comm. Protocols with Error Conditions

There is a *reinforcement* from Error Conditions to Communication Protocols. Not all the Error Conditions specified in the regulations are mandatory. However, when an optional error condition is present in the game, *e.g.* because a driver allows for these errors to be detected, it must be analyzed whether the error condition must be notified or not. This means that during the development of new versions of the Game, when new error conditions are present, the notification behavior related to the Communication Protocols should be revisited to ensure that the new information is properly reported (or consciously avoided).

Design Decisions In order to cope with *reinforcement*, we need to express the following design decisions:

- Error Condition aspect must capture the joinpoints where the error conditions are issued.
- It is necessary to perform some kind of check that ensures the error condition is notified or it is intentionally left aside for each communication protocol.

Dependency of Communication Protocols on Meters

The communication protocols access the values stored and maintained by the Metering aspect in order to report them to the monitoring systems. Consequently, communication protocols require the Metering aspect to operate as expected. When a communication protocol is enabled, meters must be present and properly fed. We need to document this dependency to ensure the consistent behavior of the system, avoiding situations where communication protocols can not accomplish their responsibilities due to the absence of meters.

Concern Execution Order

The order in which the aspects must execute can be derived from the regulations. It requires that all information that is reported must be first persisted in the SM before being made visible to the outside world. This means that after the occurrence of a relevant event in Game, such as a *play*, the expected order for aspect execution is the following:

1. Metering
2. GameRecall
3. Program Resumption
4. Error Conditions (if applicable)
5. Communications protocols (G2S or SCP).

The design documentation should make the proposed order clear even when aspectual behavior needs to be applied at different joinpoints.

To summarize, we presented some general strategies which could be used to resolve the interactions. These are the design decisions that need to be documented. In the next sections, we evaluate the Theme/UML and WEAVR aspect oriented modeling approaches to assess their ability for expressing these kinds of decisions.

5.3 Evaluation of Theme/UML

Theme/UML is the second half of the Theme approach for Aspect-Oriented requirements analysis and design. Recall that in Sect. 4.3.1 we discussed The first half, called Theme/Doc. Theme provides a process for transforming requirements in Theme/Doc into a design in Theme/UML, and moreover, claims to have support for conflict resolution. We therefore chose to evaluate Theme for our development effort. In the requirements engineering phase we have evaluated Theme/Doc, and now continue with an evaluation of Theme/UML.

The Theme/UML approach [26] is an extension of UML that provides both a notation and a methodology for modeling AO systems. In Theme/UML, a *theme* refers to a concern. A theme can consist of class diagrams, sequence diagrams and state diagrams, each of which is extended with the required notation to be able to express Aspect-Oriented concepts. Each theme is designed separately, and subsequently the themes are composed with each other. This is performed using composition relationships that detail how this is performed.

Themes are divided into two classes: base and crosscutting themes. Base themes describe a concern of the system that has no crosscutting behavior. Base themes are composed, both structurally and behaviorally, to form the base model. If a given concept appears in multiple themes, the composition can merge the various occurrences into one entity. Crosscutting themes describe behavior that should be triggered as the result of the execution of some behavior in the base model. They are designed similarly to base themes, and are parameterizable. Parameters provide a point for the attachment of the crosscutting behavior to the base model. By binding them to values of the base themes, the crosscutting themes are composed with the base model. Crosscutting themes are composed one by one with the base themes until the complete design is produced.

In accordance to Fig. 5.1, we modeled Game as a base theme and Demo, G2S, Meters, ErrorConditions, ProgramResumption, GameRecall and SCP as crosscutting themes. We found it to be straightforward to express where to attach the crosscutting behavior, both on the base themes and on other crosscutting themes. However, when considering interactions we find that Theme/UML does not perform as well. We now discuss the obstacles we encountered classified in the four different kinds of interactions (see Sect. 2.2): conflict, mutex, reinforcement and dependency.

5.3.1 Conflict

Theme/UML provides support for conflict resolution when composing different themes. These composition conflicts arise when the same diagram element in different themes has an attribute with different values. An example of this is an instance variable with different visibility specifications. Conflict resolution then consists of choosing which of the conflicting attributes to use in the composition.

The conflicts we are facing are however of a different nature. For example, consider the Demo aspect. As mentioned in Sect. 5.2.2, when it is active all conflicting aspects must be somehow deactivated. We therefore need to model the predominant nature of this aspect in some way. There is however no explicit means in Theme/UML to declare this kind of predominance. Instead, we are required to design a conflict management strategy, as explained in Sect. 5.2.2, making the conflict implicit.

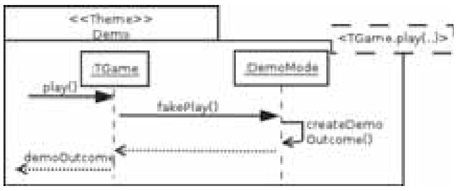
Depending on the aspect language used in the implementation, such a conflict management strategy can be realized in different ways:

1. The Demo aspect could intercept and skip the behavior of the conflicting aspects (using `around` advice without a `proceed`).
2. When the Demo aspect is deployed, conflicting aspects must be undeployed. That is, if run-time deployment and undeployment of aspects is possible.
3. If load time weaving is available, different configurations of active aspects could be loaded when the SM boots. One of these would have Demo installed, and the conflicting aspects not, a second configuration would be the inverse.

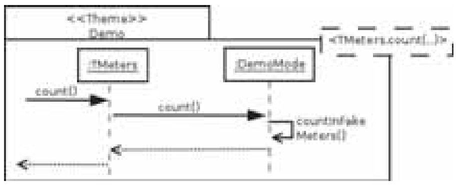
We were obliged to model the conflict management strategy using the first option. This is because, to the best of our knowledge, there is no way to fully express the other options using Theme/UML. The Theme/UML book [26] only provides a partial solution: select some concerns to form an application from a larger set of concerns that gives support to a product family. This allows *different versions* of the software to contain different configurations of active concerns. However, in our case *the same application* needs different concerns to be active in different scenarios.

We therefore model conflict management as follows: the Demo theme crosscuts the Game theme, capturing the execution of `play()` for the `Game` class. When active, Demo skips the execution of the original `play()` and instead generates a predetermined outcome (which is the main responsibility of the Demo mode). In order to keep the meters unharmed, parts of the Metering theme behavior is captured and skipped. Considering the communication protocols, their original behavior is altered: instead of responding to queries, failure (out of service) responses are returned.

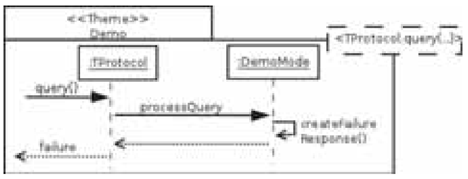
Our model is shown in Fig. 5.2. We use Theme/UML sequence diagrams, a straightforward extension of UML sequence diagrams. The figure shows three



(a) Demo on Game



(b) Demo on Meters



(c) Demo on Protocols

Figure 5.2: The Demo theme affecting the behavior defined in Game, Metering and Protocols

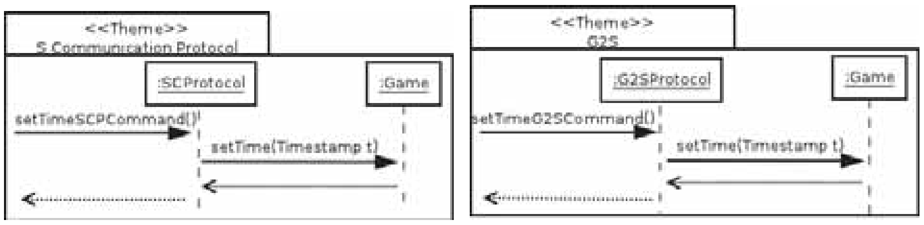
Themes, each of which has a template parameter in the top right corner, corresponding to the message send that starts the sequence. At composition time, this parameter is bound to a specific message send in the base theme, *i.e.,* the join point in the base code is identified. Also, within a sequence diagram, the behavior of the join point which is matched can be invoked. Put differently, Theme has an equivalent of the AspectJ proceed construct. The syntax to express this call is _do_*templateOperation*. Note that absence of such a call implies that the original behavior never occurs. For instance, in Fig. 5.2 there are no _do_play, _do_count or _do_query calls, which means the join point behavior is skipped.

The above proposed solution has two downsides. Firstly and most importantly, the design does not explicitly reveal the intention: the conflict between Demo and Meters, and Demo and the communication protocols. Instead it must be deduced from the implementation proposed in the diagrams. Secondly, we cannot model the conflict resolution strategy differently. Of the three design choices we proposed above, only the first could be modeled in Theme.

5.3.2 Mutex

Part of the behavior of the communication protocols is configuration command processing, as these game parameters can be set by the servers. Both protocols implement this feature, but it is not permitted to have multiple protocols setting the same value during a run of the program. The interaction we thus want to model is mutual exclusion between configuration actions: two protocols cannot configure the same item during a given program execution.

Concretely, the protocols are each modeled as a theme, where each theme defines the behavior through a set of sequence diagrams. Considering the sequence diagrams in Fig. 5.3 for the two different protocols, what we need to document is that the behavior in diagrams a) and b) cannot happen in the same program execution. However, to the best of our knowledge, Theme does not provide any



(a) SCP theme (b) G2S theme

Figure 5.3: Two themes configuring the same item in the Game.

way in which we can express this mutex relationship between both sequence diagrams. We also see no alternative solution in the same spirit as the design of the conflict interaction. Consequently, we are not able to express this mutex in the design.

5.3.3 Reinforcement

The error condition aspect reinforces the behavior of the communication protocols, reporting error conditions to the remote servers. Considering this interaction, we have a situation similar to mutex: We model the communication protocol concern as a theme, and the error conditions concern as a theme. However, we are unaware of a way in which to explicitly state the reinforcement semantics. In this particular case, we are able to integrate the reinforcement into the design, but at the cost of making the reinforcement implicit. We show this next.

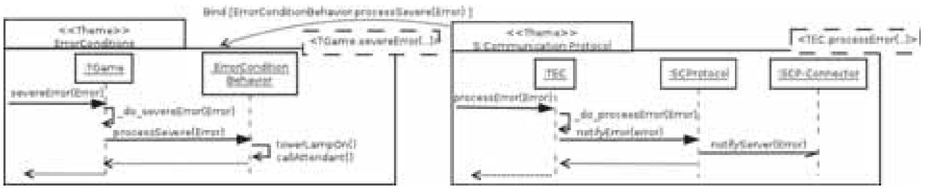
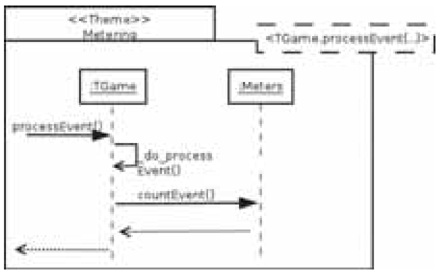


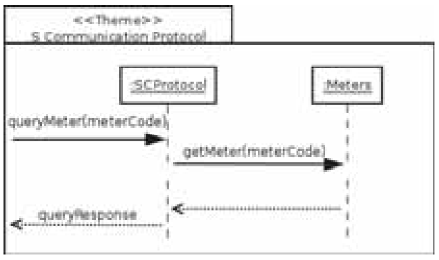
Figure 5.4: SCP Theme reinforced by Error Conditions theme.

The left hand side of Fig. 5.4 shows a sequence diagram for the most severe type of error condition. It specifies how the error event occurring causes the tower lamp to be lit and the attendant to be called. Reporting the error to the server is specified in the right hand side of Fig. 5.4 using a theme for the communication protocol. By binding both themes using the arrow construct, we define a crosscutting behavior of the communication protocol, specifying that it intercepts all calls of ErrorConditionBehavior.processSevere(Error).

However, as this states that the relationship between them is a typical crosscutting relationship, the reinforcement semantics is lost. Even though the generic behavior of the communication protocols captures all error conditions of this type, it is not clear that we know there may be new types of error conditions in the future, and each of them needs to trigger protocol notification behavior.



(a) Metering theme acquiring information regarding events on Game



(b) Metering theme acquiring information regarding events on Game

Figure 5.5: SCP theme using information acquired by Metering behavior.

This information is crucial to check the consistency of the system during maintenance and evolution. As the reinforcement semantics remains implicit here, this verification step might be omitted.

5.3.4 Dependency

The metering theme keeps track of given events in the game by changing the values of meters objects. Complementary to this, the communication protocol themes specify that to respond to queries sent by the remote server, the information stored in the meters objects is used. It is clear that the latter behavior requires the former, hence the communication theme depends on the meters theme.

The Theme/UML methodology however states that each theme defines all structure and behavior needed to provide the desired functionality, *i.e.,* in a standalone fashion. Furthermore, the designer may choose a subset of all themes to compose a system [26]. In our case this will lead to errors, as selecting the theme of a communication protocol without adding the theme of meters leads to an inconsistent design of the system. Figure 5.5 demonstrates how the meters theme gets data from the game and how this date is used later by the SCP protocol for responding to query commands.

What we need is a way to express that the meters theme is necessary whenever the communication protocol themes are composed into the system, but we have found no way to specify this in Theme/UML. Hence, we are unable to include the dependency in the design.

5.3.5 Scalability

An important feature of the AOM we require is support for scalability. As discussed in Sect. 5.1.2 we need to be able to abstract over common patterns in the design, in this case in the different themes. We however only found one mechanism that allows for such abstraction: template parameters for crosscutting templates. We now illustrate how it only addresses some of our scalability issues, using two of the examples we have seen above.

When producing the complete design, the error conditions theme shown in Fig. 5.4 needs to be bound to the occurrence of errors in a base theme. This binding expression can be a list of methods, and may use wildcards as well. As such, this one theme is an abstraction over all events in the base code that trigger a severe error. Note that if this theme would be a base theme, there would be no template instantiation, and therefore we would need to manually produce a diagram for all events that produce a severe error.

The second example of a need for abstraction is found in the specification of the communication protocols in Fig. 5.3. The diagrams for both themes are the same, save for the initiating method call and the name of the protocol class. We need to produce such duplicate diagrams for a large amount of configuration setting functionality, as both protocols provide largely the same features. Since these themes are not crosscutting there is however no template functionality available, hence we must duplicate the work.

Our only recourse for these is to have the themes as generic diagrams that the programmer needs to apply to a concrete case, refining when needed. As Theme provides no support for this, an ad-hoc solution will need to be created to guide the programmer in this effort, which is what we want to avoid, as outlined in Sect. 5.1.1.

To summarize, the composition of crosscutting themes with the base themes gives us a means of abstraction, but it does not address all our needs, and therefore Theme falls short in this respect.

5.3.6 Conclusion: Theme/UML

We found that Theme/UML does **not** allow us to express **any** of the four types of interactions in an explicit way. At the most, we are able to integrate support for conflict resolution and reinforcement into the design. However this comes at the cost of obscuring the explicit relationship between different aspects, which is likely to lead to errors during maintenance or evolution. Regarding scalability, Theme/UML allowed us to express the crosscutting relationships through templates and bindings. However bindings require a very precise information regarding where they crosscut, which is at one hand it is useful to have a clear scope of themes but, on the other hand, it is tedious and particularly difficult to maintain due to its susceptibleness to the fragile pointcut problem [80]. As a result, we consider Theme/UML inappropriate to specify the design of a SM.

5.4 Evaluation of WEAVR

WEAVR is an add-in extension to the MDE (model driven engineering) tool suite used by Motorola, adding support for AOM to their process of building telecom software [28, 29]. Morotola makes extensive use of MDE for its telecom software. Next to a UML notation, the Motorola tool suite also uses SDL [47] transition oriented state machines as the graphical formalism to define behavior. These state machines are unambiguous and allow for introducing pieces of code. This enables code generation of the complete application in C and C++. As WEAVR is arguably the best-known industrial application of AOM, with claimed support for interactions, we chose it as the second candidate for evaluation.

MDE tools have been extended to support aspect oriented concepts: aspects, pointcut, advice and some interesting/particular relationships between them. In the case of WEAVR pointcut notation is based on state machines, allowing to capture *action* and *transition* joinpoints. Wildcards are allowed to refer multiple states or actions. Advices are also expressed as state machines which are related to the pointcuts using the `bind` relationship. The WEAVR pointcut notation is based on state machines, permitting the capture of *action* and *transition* joinpoints. Wildcards are allowed to refer to multiple states or actions. Advice are also expressed as state machines, and are related to the pointcuts using the `bind` relationship. WEAVR is an aspect weaver: it combines an aspectual state machine with a base state machine when there is a join point match. The tool allows to visualize the new composed state machine, so that engineers can verify the composition for correctness before actual code generation.

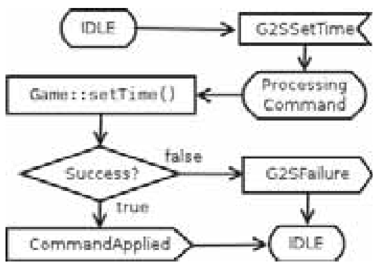
Note that although WEAVR can be used to generate the code of the application, we do not require this, we only want to specify the design. Also, due to licensing issues we were not able to use the tool for our evaluation, instead relying on published work [28, 29]. Lastly, even though SDL is a standard, the notation of its usage by WEAVR is not consistent among all the publications. The diagrams in this text are our best effort to produce a consistent notation, but we are not able to guarantee their notational correctness.

5.4.1 Conflicts

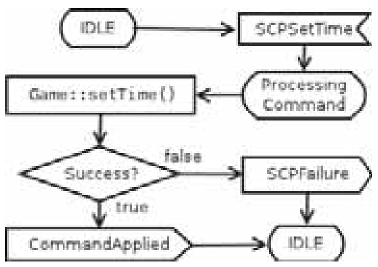
Support for conflict resolution in WEAVR is realized by the hidden_by stereotype that is used in the deployment diagrams, where aspects are applied to classes. The hidden_by stereotype relates two different aspects that intercept the same join point. The relationship states that the aspect that is hidden does not apply in those cases. For example, specifying AspectA hidden_by AspectB denotes that at a join point captured by both aspects, only the behavior of AspectB will be executed. In other words, we can state that the presence of one aspect implies the absence of another aspect, but only at the level of join points.

In our case such conflict resolution is however not sufficient as we are faced with aspects that conflict when active on different join points. For example, consider Demo: when it is active, the different protocols must return a failure (*out of service*) message upon a query of the server, which is a different join point than starting a *play*. We require instead of a hidden_by semantics that works at join point level, a similar semantics at the system or aspect level. That is, the activity of Demo should imply the inactivity of G2S and SCP.

Similar to the workaround for Theme we proposed in Sect. 5.3.1, we can provide a design that incorporates the required conflict resolution behavior. Advices in WEAVR are always around advice, and use a proceed call. As in Fig. 5.2, we can specify an around advice that intercepts Meters and the communication protocols, without performing the original behavior of the intercepted call. This workaround consequently suffers from the same drawbacks as in Sect. 5.3.1, most importantly the loss of the explicit conflict specification. The same strategy can be used to avoid other conflict behavior, such as that belonging to Program Resumption or Game Recall.



(a) State machine for G2S.



(b) State machine for SCP.

Figure 5.6: Mutually exclusive state machines for the setTime command.

5.4.2 Mutex

Recall that our mutual exclusion consists of the prohibition that, in a single run of the game, the same configuration item is configured by multiple protocols. As an example, Fig. 5.6 shows the design of the SetTime functionality for both protocols in WEAVR. The mutual exclusion in this case boils down to preventing that the state machine of Fig. 5.6a executes if the state machine of Fig. 5.6b was previously run, and vice-versa. However, WEAVR does not provide for any way in which this can be specified.

It is feasible to produce a design document that implements the mutex, but at the cost of making the explicit information of the mutex implicit. We can manually combine the different state machines for the different protocols such that the mutex relation is implemented. Briefly put, for each configuration action we combine the two state machines of the different protocols into one state machine. This combined machine contains the functionality of both protocols together with the logic that ensures that once the item has been configured by one protocol it cannot be configured by the other.

The downside of this solution is that it adds a considerable amount of tedious work, combining the state machines for all configuration settings, and obscures the intent of the design. Moreover, it produces a design where both protocols are tightly coupled. Consequently, we consider this option unfeasible and discard it.

5.4.3 Reinforcement

The design of the reinforcement from error conditions to communication protocols is similar to the design in Theme/UML discussed in Section 5.3.3. We have an error conditions aspect that handles the different types of errors that occur, and the communications protocols report these errors by intercepting this. They define a pointcut that matches on the processing of the error, and the advice then sends the corresponding notification to the server. We have however not found a means to denote the reinforcement relationship as such.

As in Section 5.3.3, the downside of this is that the explicit reinforcement relationship has become implicit, which may lead to inconsistencies during maintenance and evolution, *e.g.* when new types of errors are added to the system. An upside of using WEAVR is that its model simulation capabilities allow for consistency checking of the composed models. This could corroborate the whole execution path from the occurrence of a new error condition to the final notification to the server. However, the need for such a verification for all types of error conditions still has to be specified in the design document, and we are unaware of a means to express this in WEAVR.

5.4.4 Dependency

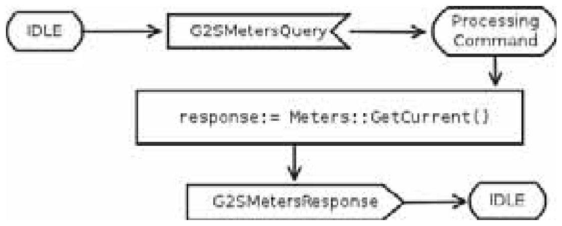


Figure 5.7: Part of the G2S Protocol state machine depending on meters.

Similar to the design in Theme, shown in Sect. 5.3.4, we have an interplay between the metering concern and the communication concerns. The metering concern captures events regarding game activity and updating the meters, while the communication protocols consult data contained in these meters when processing server requests. In Fig. 5.7, we show the latter, for the G2S protocol. The action code response := Meters::GetCurrent() refers to data previously stored in the Meters object by the Metering aspect (which is not included in the figure for clarity of the discussion). The communication protocols thus depend on the meters to provide correct functionality. Put differently, if the Meters object is available but for some reason the behavior of the metering aspect is not executed, the data returned will be inconsistent.

To declare dependency relationships, WEAVR provides for the depends_on relationship. It states that one aspect depends on another to be able to provide the required functionality. As in the hidden_by stereotype relationship, this however only applies at the join point level. If AspectA depends_on AspectB, for each shared join point the advice of AspectB will be executed before the advice of AspectA. Additionally, if AspectB does not match a join point matched by AspectA, an error is produced.

In our case however, the contact point between two aspects is the existence of the Meters object, not a shared join point. As a consequence, the depends_on relationship does not allow us to express the required dependency. This is as the semantics of the depends_on relationship is too fine grained, and only works at joinpoint level. In our case, we need to be able to express this relation at the level of aspect deployment, *e.g.* state that the deployment of AspectA implies the deployment of AspectB. WEAVR does not provide any other dependency construct, and we are not aware of an alternative option to relate the state diagrams above. We are therefore unable to include the dependency specification in the design.

WEAVR is intended to generate the code of the designed system. Its MDE approach should make the system more maintainable. However, we had no access to the tool so we could not validate this. It is a matter of further investigation to perform an experiment using the tool to generate (part of) the code of SM. The crosscutting mechanisms of WEAVR makes it resistant to the syntactic changes in the base concern, so the approach can be considered more tolerant to the fragile pointcut problem, in comparison with Theme/UML. Finally, even though the dependency and conflict support is not applicable to our case, it scales for the cases foreseen by WEAVR authors, as the interaction directives work automatically for all the interferences of the aspects.

5.4.5 Conclusion: WEAVR

We have seen that WEAVR does **not** allow us to explicitly express **any** of the four interaction types. If we allow making the explicit relations implicit, we can include support for conflict resolution and mutual exclusion in the design, the latter of which would be a large amount of tedious work. Such implicit relations however come at a cost of probable errors during maintenance or evolution. Consequently, we consider WEAVR unsuited to specify the design of an SM.

5.5 Conclusions

The AOSD-Europe technical report on interactions [74] classifies interactions in four types: dependency, conflict, mutex and reinforcement. In our software for a Slot Machine all four types are present, and we evaluated the abilities of two mature AOM approaches: Theme/UML and WEAVR, to explicitly communicate these in the design.

The somewhat surprising result of this chapter is that neither Theme/UML nor WEAVR allow us to satisfactorily express any of the four types of dependency. Although both approaches are considered mature, are accepted by the community, and furthermore claim to have support for specific kinds of interactions, it was not possible to apply them to document interactions design to our satisfaction. In our experience, their support is however at the wrong level of granularity and scope to be useful to us. In both methodologies the support is too fine-grained and the scope is too restricted.

As an alternative approach, instead of explicitly specifying the interactions, we have been able to include ad hoc, implicit support for interactions in the design. In Theme/UML, we were able to incorporate conflict and reinforcement in the design, while in WEAVR we could include conflict and mutex. However, having these relations implicit instead of explicit makes it likely for errors to arise in later maintenance and evolution phases. As a consequence, we need to discard these solutions as well.

Scalability is also weak point for these approaches, especially for Theme/UML, which require very detailed models of the system, making it hard to scale. WEAVR also requires detailed models, but its MDE nature and code generation should be a pay-off on this regard. Furthermore Theme/UML is very susceptible to fragile pointcut problem. In contrast, the pointcut notation of WEAVR, based on state transitions, makes it more resistant to this problem.

Our case study have proven to be complex enough to highlight deficiencies in current AOM approaches. We believe it is a solid test case for other AOM tools, not only for the evaluation of their interaction support but also for complex crosscutting relationships. Besides this, the modelling of interactions using ad-hoc support is another contribution of this chapter.

Having (roughly) documented the interactions, we present in the next chapter the results of the implementation.

Chapter 6

Interactions in Implementation

Part of this chapter is inspired on our previously published work: *A Fine Grained Aspect Coordination Mechanism* [100].

In Chapter 4, we have identified interactions and documented them during requirements engineering. Next, during the design phase, in Chapter 5, decisions were made regarding the software components that will form the system. These decisions include which of these will be objects and which aspects, and how they are supposed to interact. After understanding how the aspects should interact, the implementation of the final system, using aspect oriented languages, is the logical next step. In this chapter, we report on the results of such implementation.

Interaction resolution strategies, which originated in the requirements engineering phase, and were expressed in the design documents, can be implemented in different ways, according to the features offered by the underlying programming language. Therefore, we found it necessary to assess the consequences generated by the selection of a static or a dynamic aspect language.

In the following sections, we compare the implementation of the interactions using static and dynamic aspect oriented languages. Using this experience, we evaluate the impact that the different languages have on the implementation of these interactions. For example, the availability of run-time weaving in dynamic languages impacts directly on the implementation of the *conflict* resolution strategy. We then discuss different improvements, considering our implementation, in order to provide better separation of concerns and ease the maintenance of the aspects and the implementation of their interactions.

6.1 Static and Dynamic AOP Languages

AOP languages [34, 52, 78] usually are extensions to existing programming languages [14, 38, 42, 45], as they add new features to a programming language. AOP languages enhance a base programming language with the *aspect* module which contains its behavior in the form of advice. The aspect also defines pointcut designators, which capture the joinpoints in the base program. For example, consider AspectJ [52], which is an aspect oriented extension of Java [42]. AspectJ provides notations for aspects, advice and pointcuts on top of Java.

AspectJ is the pioneer aspect language, and a lot of research has been carried out on it. It is, arguably, the most mature and widely used aspect oriented language. It has been used as part of an industriallyy accepted middleware framework for Java, called Spring framework [79]. Furthermore, it has mature development tools, such as the AJDT plugin for the Eclipse IDE. For these reasons we have chosen AspectJ as one of the languages to study the implementation of aspectual interactions.

A key distinction between (aspect oriented) programming languages is their dynamic or static nature. Java and therefore AspectJ are static. Weaving is performed at compile or load time, but definitively, **before** run time. Once aspects have been woven, they remain as part of the system. Aspects, in AspectJ, cannot be uninstalled nor removed by any means. In addition, once the system is running, it is not possible to deploy new aspects.

On the other hand, dynamic programming languages allows for modifying and extending programs at run-time. Hence, they provide the necessary tools for the implementation of run-time weaving of aspects. Therefore, dynamic aspect languages allows for run-time deployment and removal of aspects.

PHANtom [34] is an aspect oriented language extension for Smalltalk, arguably the most influential object oriented language. PHANtom allows for fine grained control on the application of aspects. It supports advanced aspect scoping mechanisms such as *computational membranes* [83], reentrancy control, and dynamic ordering for aspect application. These features make PHANtom a good example of a general purpose dynamic aspect oriented language. Therefore, we decided to also implement the slot machines software using PHANtom to compare its results against the implementation in AspectJ.

We choose to implement our interactions using both a dynamic and a static language, in order to compare the results obtained for each scenario. The comparison is not oriented towards obvious topics such as performance. We will not discuss the overhead that dynamic aspect languages may impose, neither the lack of flexibility of static languages. Instead, our comparison is aimed at evaluating how different languages features impact on the final implementation of the interactions, and which are the pros and cons of the mentioned implementations, fundamentally, with respect to maintenance.

Regardless of their dynamic or static nature, aspect oriented languages provide little or no support for aspect dependencies and interactions. When provided, interactions support is usually aimed at working at joinpoint level, as reported by Cleenewerck *et al.* in [88].

Due to the particular semantics of the interactions in the SM domain and the poor support for interactions built in the languages used, we were forced to implement the interactions in an ad-hoc manner, in a similar way as we roughly modeled interactions during design phase due to limitations of the existing approaches.

In order to evaluate the effect stemming from the use of these languages, two independent implementations of the SM were carried out. These were done by two different engineers with knowledge of the domain in order to avoid learning effects. This allows us to more objectively evaluate the impact of the static or dynamic languages for the same interaction types on the same domain. In both cases, the engineers counted with extensive experience in the development of software for SM, which ensures familiarity with the vocabulary used in requirements and design documentation, and a similar degree of understanding of the system.

To ensure a common base line for the development, the following actions were carried out:

- Both engineers were provided with the same requirement documents.
- They knew about the concern decomposition chosen (presented in Sect. 3.7).
- They were instructed regarding the interaction types proposed by Sanen *et al.* in [74].
- The interaction instances found in the domain were introduced to them as in Sect. 3.8.
- The general idea regarding where crosscutting concerns should cut the base (Game) concern was presented, as explained in Sect. 5.2.1.
- The design decisions on the strategies for treating the interactions were presented, as explained in Sect. 5.2.2.

Note that no detailed design was provided, therefore there are some minor and incidental differences in the resulting implementations.

6.2 Implementing Interactions in PHANtom

PHANtom [34] provides typical aspect oriented features (aspect, advice, pointcuts) as well as advanced mechanisms based on the powerful Smalltalk reflective capabilities (e.g. computational membranes). Therefore, there are many possible ways for implementing the interactions that we analyze along this work. In order to make the implementations in AspectJ and PHANtom comparable, we decided not to use the most advanced mechanisms available in PHANtom. Instead, we applied a subset of PHANtom's features that can be found in most aspect oriented languages. Being that PHANtom is an extension of a dynamic language, we took advantage of the possibility of run-time deployment of aspects.

Fig. 6.1 presents main classes and aspects for the PHANtom based implementation. This design has been developed by the implementor of the PHANtom version of the SM software. The notation is an ad-hoc variant of UML class diagram where each package box represent a concern and a crosscutting relationship is denoted by a dotted line with the corresponding stereotype. Our PHANtom based implementation consists of the following concerns and modules (classes and aspects):

Game concern implemented in several classes, of which the more relevant are: `Game` and `OutcomeGenerator` (in charge of generating the random output for each *play*).

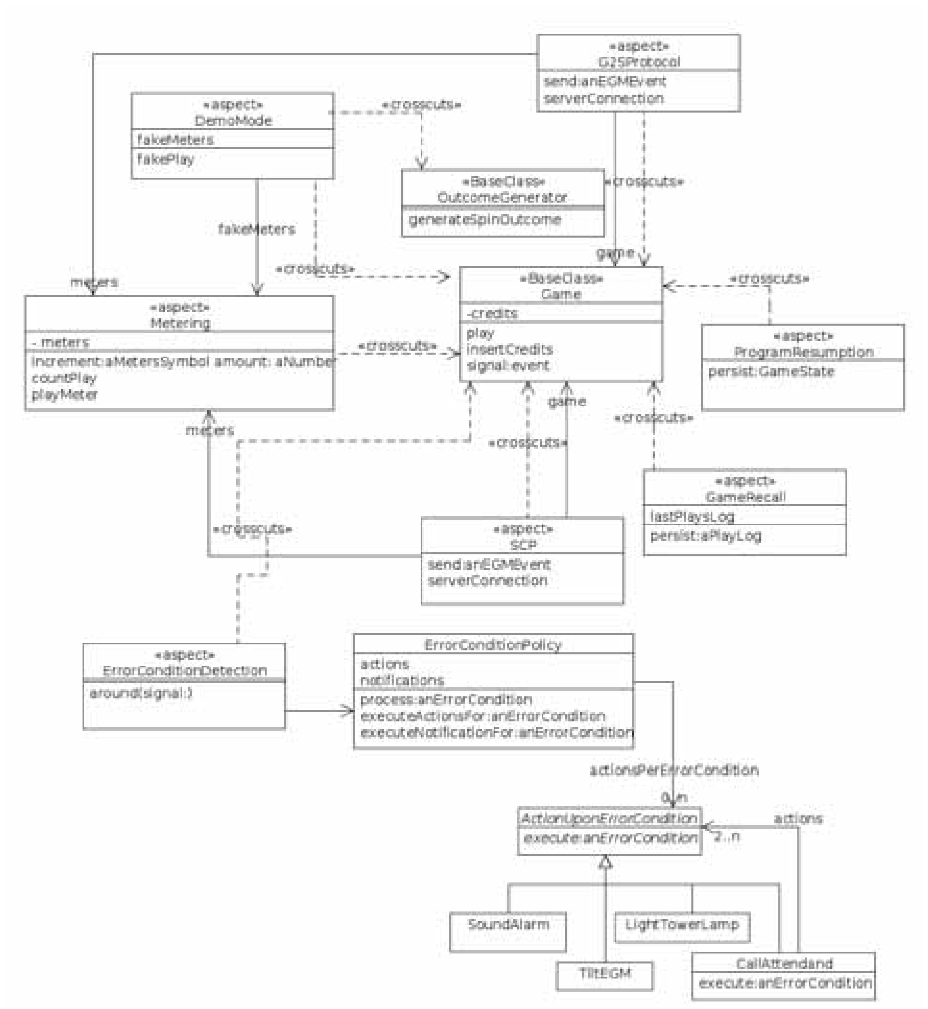


Figure 6.1: Classes and aspects for the PHANtom based implementation.

Game Recall concern implemented mainly in the GameRecall aspect and some helper classes (such as a log of the last *plays*). GameRecall aspect captures relevant information for each play, packages it and stores it using the persistence services provided by the platform.

Error Conditions concern implemented as an aspect capturing the occurrence of events that require some specific action(s) to be carried out. There is a hierarchy of objects representing the different available actions. Consider for example, the event of the *printer* running out of paper. In this case it is necessary to light the tower lamp. For other events, more than one action may be needed. Therefore, several actions can be associated with an error condition. If no action is associated, the detected error condition is silently ignored.

Demo Mode concern implemented in the `Demo` aspect whose main responsibility is to allow the selection of the next awarded prize. It adds a menu for the selection of the prize and it replaces, using an `around` advice, the behavior of the `OutcomeGenerator` to effectively simulate the awarding of the selected prize.

Meters concern There is a `Meters` class, which uses a dictionary for storing the different meters. The Metering aspect crosscuts the Game concern objects to obtain the relevant information that needs to be counted.

SCP concern This communication protocol concern is mostly implemented in the SCP aspect, which crosscuts the game in order to report the SM activity to the monitoring system. This concern also includes hierarchies of events (objects that encapsulate information flowing from the SM to the monitoring system) and commands (objects containing configuration information sent from the monitoring system to the SM).

G2S concern This is very similar to SCP. The G2S aspect contains most of the logic for this protocol. Events and commands used by this concern are shared with SCP.

Note that most event and command classes are shared by the two communication protocol concerns. G2S and SCP differ mainly in the low level serialization format of messages (events, commands), and in the frequency and pattern of message exchange and the events that triggers these exchanges. For example, G2S allows the monitoring system to register to certain meters and get their value at regular time intervals, while SCP works only by polling. This is why *events* and *commands* can be shared, but the logic of the protocols is independent.

6.2.1 Dependency

As explained earlier in Sect. 5.2.2 and documented in Sect. 5.4.4, the Metering aspect is needed by the communication protocols (G2S and SCP), in order to report the SM state.

For this implementation, the `G2S` and `SCP` aspects each have a reference to the `Metering` aspect. The Metering aspect instance to be used during a run is passed as a parameter to the communication protocol in the corresponding constructor. Note that `G2S` and `SCP` aspects implement only one constructor, which requires the `Metering` aspect instance (see Listing 6.1), since it makes no sense to instantiate a communication protocol without a meters source. Hence dependency is enforced as a side effect of the instantiation order of our aspects, and the constructors that must be used for the protocols.

The protocols report what the `Meters` object knows about the SM state. To make certain of these values are updated, it is required that Metering aspect is installed. In turn, to ensure that `Metering` aspect is deployed, hence the dependency is satisfied, communications protocols should invoke the `isInstalled` method. This is a message define in the `PhAspect` class (PHANtoms's superclass for all aspects), and that returns true if the receiver (an aspect instance) has been woven.

This run-time check plus PHANtoms's ability of doing dynamic aspect deployment allows the dependent to programmatically install its dependencies.

That is, communications protocols instances can install the `Metering` aspect if they find this has not yet been done .

From this, we can conclude that, in PHANtom, honoring references between aspects is not enough to guarantee that the dependency is satisfied and that the system will behave as expected. Extra logic for checking required aspects installation is needed, but this code gets buried with the rest of the dependent aspect code (in this case in the `install` method, see `install` method in Listing 6.1), making this information implicit. The lack of explicit support for dependency make the maitenance of this relationship error prone.

Listing 6.1: CommuncationProtocol abstract class constructor .

```
CommunicationProtocol class>>
    for: aGame server: aServer meters: aMetersAspect
  ^ (super new)
     server: aServer;
     meters: aMetersAspect ;
     initializeGame: aGame;
     yourself

CommunicationProtocol>> install
  self meters isInstalled
          ifFalse: [self meters install].
  ^super install.
```

6.2.2 Reinforcement

There is a reinforcement from the *error condition concern* to the communication protocols. This means that extra functionality that otherwise is not available, is enabled in the G2S and SCP communication protocols as a result of the availabity of a new error condition to be detected. More concretely, this means that a new error condition enables the communication protocol (and the corresponding monitoring system) to act accordingly. This design decision has been documentation in Sect. 5.3.3.

From the programming standpoint, implementing a reinforcement means that when the system is able to detect a new condition *it must be decided* whether it needs to be reported to the monitoring systems using the protocols. The error condition must be considered, and the notification to the monitoring system must be programmed or configured accordingly.

This is error prone, as error conditions may originate through a myriad of circumstances. Furthermore, the programmers who are aware of them usually are not familiar with regulations and technical specifications of communication protocols, which in turn define the proper behavior upon the occurrence of an error condition.

Hence, the design and implementation of the reinforcement interaction must ensure that the notification of each error condition is properly considered. In order to help in this process, our implementation forces a decision making on what to do with each error condition. If there is no explicit decision, the reinforcement ad hoc mechanism raises an error and the system is halted.

The UML class diagram in Fig. 6.2 presents the reinforcement interaction. The `ErrorConditionDetection` aspect is in charge of capturing the occur-

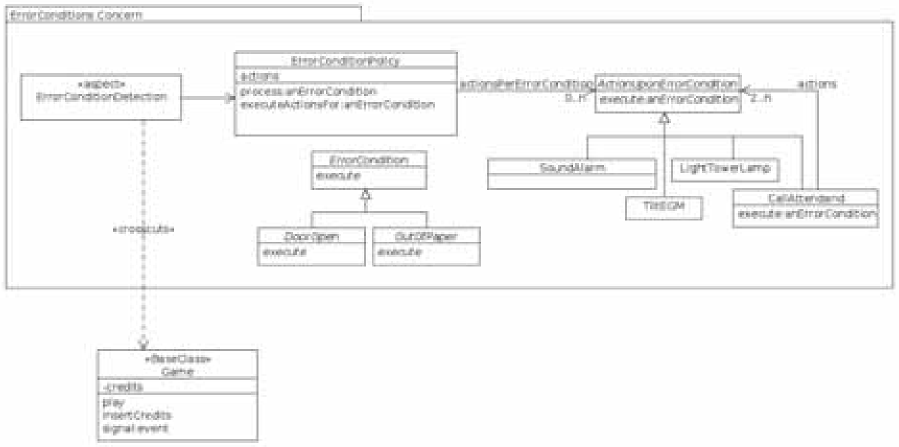


Figure 6.2: Class diagram of Error Conditions implementation in PHANtom.

rence of any *error condition*. Even though error conditions can be raised in different parts of the system, they are notified to the Game core in well known places. Once an error condition is captured, different actions can be carried out, for instance, sounding an alarm, switching the SM to *out of service* state, calling the attendant by lighting the tower lamp, etc. Therefore, there is an error condition policy object which keeps the *actions* associated with each error condition, and which executes them upon error condition capture.

The reinforcement states that once the *actions* associated with an error condition policy are executed, notifications to the communication protocols must be dispatched.

In order to keep this functionality modularized, an aspect implementing the interaction has been defined. This aspect isintroduced in this stage, as it was not present at the design phase.The `EC2CPReinforcement` (**E**rror **C**ondition to **C**ommunications **P**rotocol) aspect introduces an instance variable on the definition of the `ErrorConditionPolicy` class, as shown in Listing 6.2. This reference holds a collection of notifications to be dispatched for the associated error condition. The aspect also uses an `after` advice to send the notifications for the communication protocols on the *actions* execution. This collection of *notification* objects cannot be empty. If it is case, the system is halted with a meaningful error message. If the error condition is such that it must not be reported to the monitoring system, this can be expressed by instantiating a *NoNotification* object (an instance of the NullObject design pattern [95]).

Listing 6.2: Part of the initialize method for the reinforcement implementation aspect.

```
EC2CPReinforcement>>initialize
| modifier |
  [modifier := PhClassModifier new on: (PhPointcut receivers:
            'ErrorConditionPolicy' selectors: #any asParser).
  modifier addNewInstanceVar: 'notificationsPerErrorCondition'.
  modifier install.
```

Listing 6.3: Notification execution code in the reinforcement aspect.

```
EC2CPReinforcement>>executeNotificationsFor: anErrorCondition
 (ecPolicy  notificationsFor: anErrorCondition) isEmpty
    ifTrue: [
      self error:'Reinforcement error:notification for EC is
      missing']
    ifFalse: [
      (ecPolicy notificationsFor: anErrorCondition)
       do: [:notification |
              notification notify:anErrorCondition.
             ]]
```

This approach to the implementation of reinforcement presents the following upsides:

- It forces the programmer to make a decision on the need (or not) to notify the error condition. If a decision is not taken, the first time the offending error condition is raised, the system will halt.
- The intention of the programmer at the time of the decision is explicitly recorded in the code. If an instance of `NoNotification` is found, it means that the error condition must NOT be notified. The difference between this case and a forgotten notification is therefore obvious.
- The logic of reinforcement is decoupled from the `ErrorCondition` aspect, and encapsulated in a separate aspect. This allows for easy maintenance of both of them.

6.2.3 Conflict

As seen in Sect. 5.2.2 Demo concern must change the behavior of Game in order to allow affecting the prize awarding algorithm. Design decision on this regard has been documented in Sect. 5.3.1. To implement the core functionality of the "demo" concern, our Demo aspect uses an `around` advice in order to replace the behavior that generates the random outcome, as depicted in the Fig. 5.2a.

The *conflict* between "demo" mode and meters, communication protocols, game recall and program resumption concerns dictates that when the SM runs in "demo" mode, part of the behavior of the aspects that implement Program Resumption, GameRecall, Metering and communication protocols need to be partially or completely avoided or skipped. There are several alternatives for implementing this behavior. In this particular case, we decided to use the dynamic deployment of aspects offered by PHANtom.

During deployment of our Demo aspect, the following actions are performed to cope with the conflict (see Listings 6.4 and 6.5):

- In order to keep the Meters unharmed with *demo plays*, a reference to the real meters is kept by Demo, and the *Meters* object used by the *Metering* aspect is replaced. In this way, the meters during the demo session reflect exclusively data derived from the plays in `demo` mode. At the same time, the original meters are kept safe.

- An *around* advice instruments how the communication protocols report the SM behavior. Instead of responding according to an event occurred during the *demo* session, the SM is reported as being *out of service*. This keeps the reported information consistent, and demo activity is not considered for the accounting.

- In order to avoid persisting information regarding demo plays, the Program Resumption aspect is undeployed.

- To avoid mixing information of regular and "demo" plays in the GameRecall log, we decided to use the same strategy employed with Metering, that is, to temporarily replace the GameRecall log for a fake one. This allows the certifier to recover the log of the plays he completed without harming the original plays.

Fig. 6.3 presents a sequence diagram for the deployment of Demo, and how conflict management is implemented in this case.

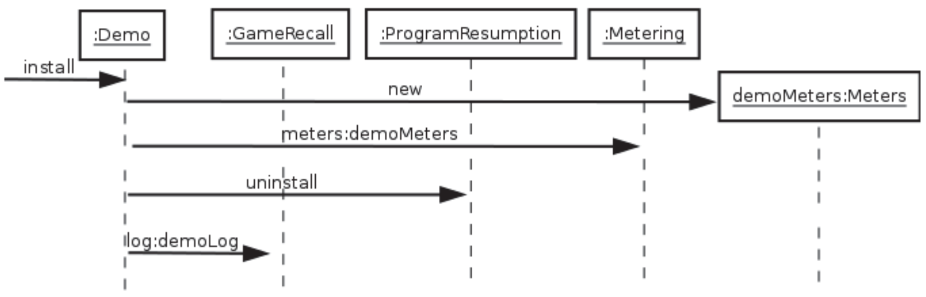


Figure 6.3: Sequence diagram for Demo aspect installation in PHANtom.

From the UI standpoint, our Demo aspect also adds extra interface items. A label indicating that demo mode is active is displayed. Besides this, there is a menu that allows the player to select the prize that must be awarded in the next play. As these elements do not interfere with other concerns, we will not explain them in further detail.

The behavior mentioned above is triggered by the *install* method of our Demo aspect (the method used to deploy aspects in PHANtom). This means that weaving Demo implies all the mentioned actions to be carried out on the system, assuring the correct behavior of the SM during Demo mode.

Listing 6.4: Part of Demo aspect initialize code.

```
Demo>>initialize
 ....
  pcGenerator := PhPointcut
    receivers: 'OutcomeGenerator'
    selectors: 'generateOutcome'
    context: #(#receiver #proceed)
    if: [ :ctx | ctx receiver = self game generator ].
  adviceGenerator := PhAdvice
    around: pcGenerator
    advice: [ :context |
      self generateFakeOutcome.
       ].
  self add: adviceGenerator.

  pcG2S := PhPointcut
    receivers: 'G2S'
    selectors: 'sendToServer:'
    context: #(#receiver #proceed)
    if: [ :ctx | ctx receiver = self g2sAspect ].
  adviceG2S := PhAdvice
    around: pcG2S
    advice: [ :context | context receiver reportOutOfService ].
  self add: adviceG2S
```

When Demo is unweaved, the around advices which replaced the outcome calculation and those that affected the communication protocols, are removed. During the un-installation of the Demo aspect the following steps are followed (see code Listing 6.5):

- The original meters are restored, and the *fake* ones discarded. This is done by restoring the original Meters instance used by the Metering aspect before the SM entered into Demo mode.
- Program Resumption is re-installed (weaved) into the system, so that persistence is enabled once again.
- Game Recall log is replaced by the original log object.
- The original behavior of communication protocols is restored as a consequence of the unweaving process.

Listing 6.5: Demo aspect install and uninstall code.

```
Demo>>install
  super install.
  originalMetersContents := metersAspect metersContent copy.
  originalLog:= recallAspect log copy.
        programResumptionAspect uninstall.
  self aspectMembrane advise: self g2sAspect aspectMembrane.

Demo>>uninstall
  super uninstall.
  metersAspect metersContent: originalMetersContents.
  recallAspect log: originalLog.
```

It could be argued that in some cases communication protocols can be deactivated by unweaving the corresponding aspects. However, the expected behavior during Demo mode is the SM remain "visible" to the monitoring system. Moreover, we preferred here to remove part of the communication protocol behavior (it does not report events to the monitoring system), to illustrate both situations: complete removal of an aspect (ProgramResumption) or partial removal (G2S).

6.2.4 Mutex

In our case study, Mutex interaction refers to the impediment of receiving configuration (writing) commands for the same configuration item from different communication protocols. Recall that this interaction could not be satisfactory documented in Chapter 5. Configurable items include payment options, such as ticket-in ticket-out settings, clock information, setting the SM out of service or restoring it back in service, progressive prize accumulator value, etc.

The interaction resolution strategy for *mutex*, as defined in Sect. 5.2.2, allows only one communication protocol **per** configuration item. This means that a configuration of different items may come from different communication protocols, but for a single item, only one protocol is allowed for configuring it. To implement this, it is necessary to establish which configuration item (or their corresponding command objects) can be set by which protocol. Commands arriving from the wrong protocol are then ignored. These occurrences are logged for further study, as they may indicate an erroneous configuration of the monitoring system servers.

Our implementation in PHANtom uses a `MutexController` aspect, which is in charge of intercepting the reception of configuration commands (using an around advice) coming from both protocols. In this particular implementation, and due to the fact that communication protocols belong to a hierarchy, the MutextController aspect captures the `#process:` method on all sub-classes of the *CommunicationProtocol* aspect, as shown in Fig. 6.4. Once a configuration command arrives, it is checked against the configuration. If it arrived from an allowed protocol, the command is applied (`proceed` is called). Otherwise, it is discarded and the occurrence is logged.

Note that in this case, the interaction has been implemented in a separate aspect. As a consequence, the SCP and G2S concrete aspects are not coupled, since they are coordinated by a third party, the `MutextController` aspect. Part of the initialize method, where the pointcut and the advice are defined is presented in Listing 6.6

Listing 6.6: MutexController initialization code.

```
1  MutexController>>initialize
2   |pc advice command |
3    super initialize.
4    pc := PhPointcut receivers: 'CommunicationProtocol+' selectors:
5    process:' context: #(#receiver #proceed #arguments).
6    advice := PhAdvice
7        around: pc
8        advice: [ :context |
9     command := context arguments at: 1.
10    (configuration at: command class ifAbsent: [ nil ]) =
11             context receiver class
12                   ifTrue: [context proceed]
13                   ifFalse: ["nothing to do, warn the programer"].
14    nil ].
15   self add: advice.
16   ...
```

Regarding the scalability when adding a new protocol, its `process:` method will be automatically intercepted (Listing 6.6 line 4), so that their configuration commands will be under the control of our MutexController. Furthermore, adding a new command does not require additional changes in the code, the command will be applicable once the corresponding *mutex* configuration is updated accordingly.

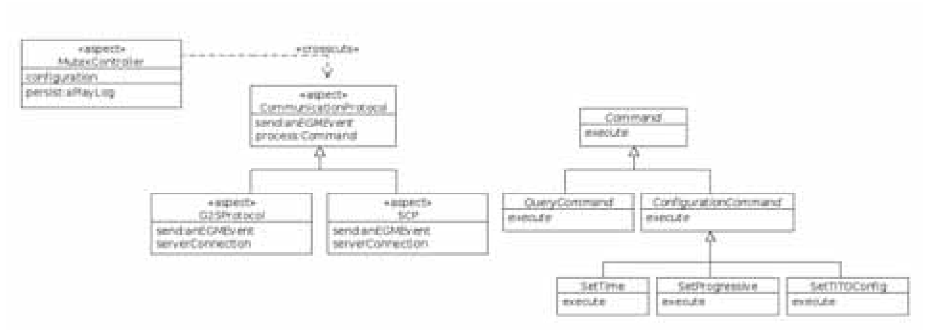


Figure 6.4: Mutex implementation in PHANtom.

6.2.5 Summary

From the implementation in PHANtom it is clear that the implementation of the interaction resolution strategies requires deep knowledge of internal details of the involved aspects. As an example, consider the *conflict*, where the programmer needs to know some implementation details to avoid mixing data coming from regular and demo plays. This adds extra coupling, as a change in the implementation of the meters concern (for example, splitting the meters into different objects), will impact in the *conflict resolution strategy* code. However, we do not consider this a downside originating from PHANtom. Instead, it seems to be an obvious need derived from the desire to control certain behaviors from our aspects without re-implementing them completely,

The dynamic features of PHANtom have an impact on conflict management. If the resolution strategy for a conflict can be implemented in terms of unweaving of aspects, it can be implemented straightforwardly, as this operation is supported by PHANtom.

On the other hand, the few compile time checks present in PHANtom require us to implement an ad-hoc mechanism to ensure reinforcement. As we will see in the following sections, where the AspectJ based implementation is presented, this mechanism can be replaced by compile checks.

For dependency and mutex there is no evident impact of the dynamic features offered by PHANtom. Dependency is implicitly implemented, through a instance variable which reference that must be provided somehow. Instead, Mutex is implemented a separate aspect encapsulating the logic to avoid more than one communication protocol to configure the same configuration item.

6.3 Implementing Interactions in AspectJ

The implementation of the SM in AspectJ follows the general design explained Sect. 5.2, where the Game concern is implemented as a set of classes which are crosscut by the aspects in charge of implementing the crosscutting behavior. An overview of the main classes and aspects of this implementation is shown in Fig. 6.5.

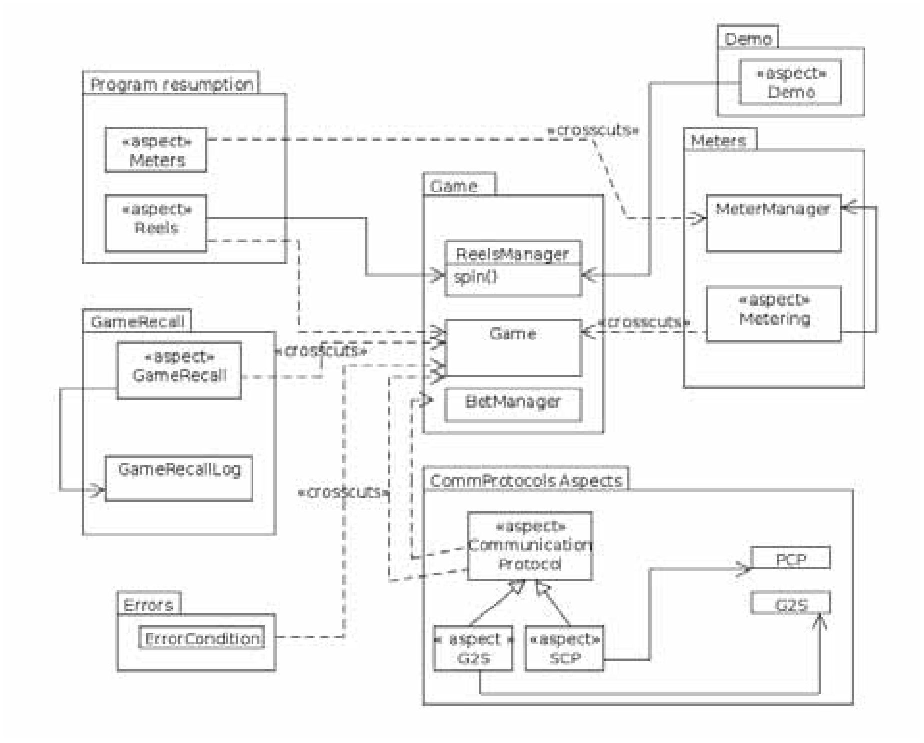


Figure 6.5: Class diagram for the SM implementation in AspectJ

Some relevant information regarding the implementation follows:

- Error conditions are organized as a hierarchy, with the `ErrorCondition` abstract class as its root. Each error condition maintains a reference to a list of *actions* objects (instances of sub-classes of the `Action` abstract class), which represent the operations that must take place upon the detection of an error condition.

- The G2S and SCP concerns are implemented as two concrete aspects, which are sub-aspects of the abstract `CommunicationProtocol` aspect.

- The Metering aspect crosscuts the Game concern objects and keeps a MetersManager which stores the value of all the meters, which is equivalent to the *Meters* class in the PHANtom based implementation.

- The GameRecall and ProgramResumption concerns are implemented as the corresponding `GameRecall` and `ProgramResumption` aspects, which crosscut Game concern objects, taking the relevant information they need to persist.

6.3.1 Dependency

For this implementation, the `MetersManager` object is in charge of keeping the value of the current meters. These values are fed by the `Metering` aspect. The communication protocols use the values stored in the `MetersManager` to respond to query messages from the monitoring system. More specifically, the `GetMeter` message, reads the value from `MetersManager`.

In this case, the dependency of the G2S and SCP concerns on the Meters concern is reified as a reference. There is as an object (the `GetMeter` command) belonging to the communication protocol concerns, which *reads data from* the `MetersManager`, belonging to the Meters concern (see Listing 6.7).

The communication protocols depend on the deployment of the Metering aspect. The AspectJ compiler ensures that aspects are correctly installed if they are included in the build configuration. This means that if a (buggy) configuration file excludes the Metering aspect, the values stored in MetersManager will be out of date.

In AspectJ it is possible to enforce that the required Metering aspect is installed by using the *aspectOf()* static method, which returns the singleton aspect (the default instantiation policy for aspects in AspetJ). This can be seen in line 2 of Listing 6.7. Alternatively, the `hasAspect()` can be use.

If the required aspect is missing in the configuration file, an *unresolved type* error is raised during compile. The down side of this approach is, once again, that (part of) the code related to the dependency is tangled with the rest of the code of the dependent aspect, making it hard to maintain.

Listing 6.7: Dependency in AspectJ implementation.

```
1 after(GetMeter event): cflow(g2sExecute()) && getMeter(event) {
2   Aspects.aspectOf(concerns.meters.aspects.Metering.class);
3   GetMetersResponse resp = new GetMetersResponse();
4   resp.setMeterName(event.getMeter());
5   resp.setMeterValue(metersManager.getValue(event.getMeter()));
6   send(resp);
7 }
```

Although this implementation is slightly different from the PHANtom based, it is equivalent in the sense that communication protocol objects (in this case the concrete command that reads data) require a reference to the meters value holder (in this case `MetersManager`). In both cases, the dependency is implicit, and the dependent aspect can check the existence of the required aspect. In the case of PHANtom it is also possible to deploy it on demand, for example, a communication protocol that calls the `install` method on a Metering aspect instance. Hence, we conclude that there is no significant difference between PHANtom and AspectJ based implementations of the dependency interaction.

In order to help the programmer to locate the bug more easily, it is necessary explicit support for dependency, and runtime checks indicating the dependency is correctly (un)satisfied. This explicit dependency support or alternatively, the ad-hoc check are important for efficient debugging. In our previous experience with SM, inconsistent meters reporting was a significant bug, and a lot of testing effort was devoted to check their consistency. Therefore, we consider a requirement for the aspect languange to be used, to allow some kind of checking ensuring the dependency is satisfied.

6.3.2 Reinforcement

The implementation of error condition execution in AspectJ is similar to that of PHANtom. In both cases there are a set of *actions* associated to the error conditions. These actions are executed when an error condition has been signaled.

Recall that the objective of the reinforcement implementation is to enable the notification of error conditions to the monitoring system through the communication protocols. In the case of the addition of a new error condition supported by the system, it is critical to ensure that the possibility of notifying monitoring systems, has been considered by the programmer. In consequence, *the implementation must supply a mechanism to force the programmer to make a decision in this regard.*

In this case, the reinforcement interaction is implemented taking advantage of static language features:

- The G2S and SCP protocol aspects use an inter-type declaration to introduce an abstract method called `notifyG2S(G2S)` or `notifySCP(SCP)` respectively, in the `ErrorCondition` abstract class. This can be seen in Listing 6.8, line 3.This forces the implementation of the corresponding concrete method in order to provide specific behavior for each error condition and for each protocol.

- The `notifySCP` and `notifyG2S` methods are defined in the respective communication protocol aspect class and introduced into the error condition classes through intertype declarations. An example for two error conditions (DoorOpen and OutOfPaper) are included in Listing 6.8, lines 5 to 10.

Listing 6.8: Reinforcement implementation for AspectJ.

```
1  public aspect G2SAspect extends CommProtocolAspect {
2    ...
3    public abstract void ErrorCondition.notifyG2s();
4
5    public void concerns.errorConditions.DoorOpen.notifyG2s() {
6      getProtocol().sendEvent(new Notification("DoorOpen"));
7    }
8    public void concerns.errorConditions.OutOfPaper.notifyG2s() {
9      getProtocol().sendEvent(new Notification("OutOfPaper"));
10   }
11    ...
```

Given a new error condition, which is added to the implementation of the the SM in AspectJ, the inherited abstract method will force the implementation (and therefore make a decision) on what to do with the new error condition. Otherwise, the compilation process will fail as there is a missing implementation for an abstract method. This is shown in Fig. 6.6 where in grey we show the places where new code is needed for this case.

An additional advantage of this implementation is that, in case of including a new communication protocol, this mechanism ensures that each error condition will be considered for its notification. This is due to a new `notifyNewProtocol()` abstract method that will be introduced by the corresponding aspect in the root of the error conditions hierarchy (the `ErrorCondition` class). This is shown in Fig. 6.7. In this way, until an implementation of the `notifyNewProtocol()` abstract method is provided for **each** error condition, the program cannot be compiled.

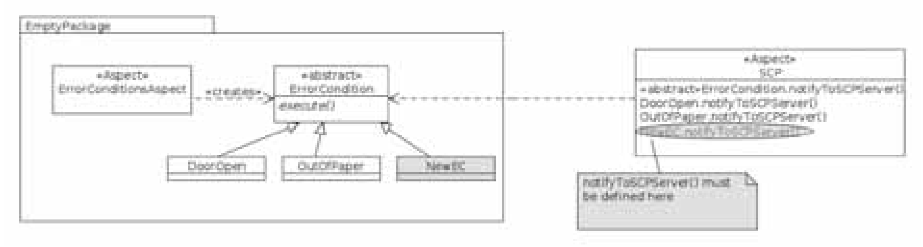

EmptyPackage
«Aspect» ErrorConditionsAspect
«creates»
«abstract» ErrorCondition
execute()
DoorOpen
OutOfPaper
NewEC
«Aspect» SCP
«abstract» ErrorCondition.notifyToSCPServer()
DoorOpen.notifyToSCPServer()
OutOfPaper.notifyToSCPServer()
NewEC.notifyToSCPServer()
notifyToSCPServer() must be defined here

Figure 6.6: Class diagram for reinforcement. In grey we mark where changes need to be made for adding a new error condition.

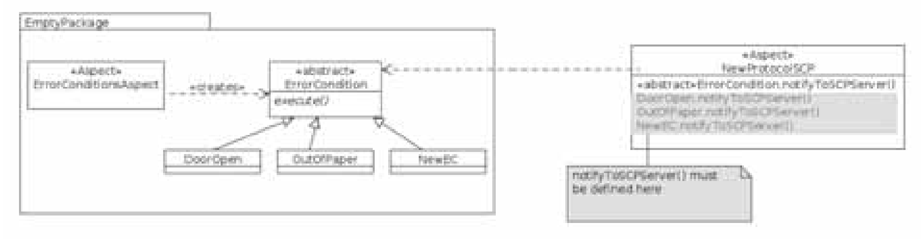

EmptyPackage
«Aspect» ErrorConditionsAspect
«creates»
«abstract» ErrorCondition
execute()
DoorOpen
OutOfPaper
NewEC
«Aspect» NewProtocolSCP
«abstract» ErrorCondition.notifyToSCPServer()
DoorOpen.notifyToSCPServer()
OutOfPaper.notifyToSCPServer()
NewEC.notifyToSCPServer()
notifyToSCPServer() must be defined here

Figure 6.7: Class diagram for reinforcement. In grey we mark where changes need to be made for adding a new communication protocol.

The implementation of the *reinforcement* from the ErrorConditions concern to the SCP and G2S concerns is implemented differently due to language characteristics. The differences between these implementations developed using PHANtom and AspectJ are significant. Due to the static nature of AspectJ, it was possible to implement the reinforcement in a way that takes advantage of the compilation process. The compiler will fail in case an error condition lacks concrete behavior defined for the supported communication protocols. In contrast, the PHANtom based implementation can detect the missed reinforcement only in run-time, the first time the error condition is raised.

6.3.3 Mutex

For *mutex* interaction, configuration commands from SCP and G2S concerns for the same configuration item must not be allowed. To implement this behavior, there is a *lock* (part of our implementation) for each configuration command, which needs to be acquired by the aspects implementing each communication protocol. In this way, a given configuration item can be configured by a command coming from one protocol only if the protocol is allowed to do that, i.e., if the lock is granted to that protocol.

The `Mutex` aspect delegates the logic to decide if a lock can be conceded to an object or not, in the form of the Strategy design pattern. This allows for different locking strategies, for example:

- FirstLockerKeepsLock: that is, the first object that obtains the lock is the only one allowed to regain it in the future. This means that if the same command arrives from a different protocol, its execution will be skipped, otherwise it will proceed. This is useful as a default policy so that each configuration item can be configured, at least by some protocol, in a consistent manner.
- ParameterizedLockStrategy: this holds a configuration that specifies which command is allowed to come from which communication protocol. The command and its source are compared against this configuration in order to determine if the execution of the command should proceed or not. This is similar to the PHANtom based implementation.

Besides the differences regarding the possibility of selecting an specific locking strategy, which is not available in our PHANtom implementation, the mutex interaction follows a different approach to that on PHANtom. In this case Mutex is implemented as an **abstract** aspect with an abstract pointcut. Each subaspect must define the concrete pointcut to match the joinpoint where mutex is needed. Typically this is the `execute()` method of configuration commands. Therefore, there will be a subclass for each command, as it is exemplified in Fig. 6.8.

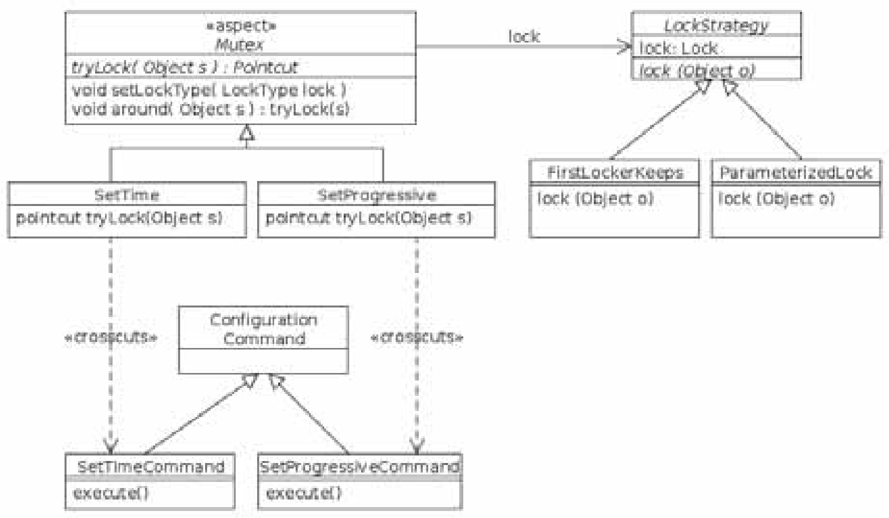


Figure 6.8: Class diagram for mutex interaction implemented in AspectJ.

Fig. 6.8 presents a class diagram for mutex. Two concrete aspects have been defined for two different configuration items (Time and Progressive) and their respective commands.

Even though the implementation in AspectJ is different from the one written in PHANtom, there are no essential differences due to the features provided by the programming languages used. Instead, they are a product of different design and implementation decisions, both of them valid and compliant with the requirements.

6.3.4 Conflicts

The implementation of *conflict* interaction for AspectJ is somewhat different from the one proposed in PHANtom. Due to AspectJ limitations, the Demo aspect can not be deployed dynamically. Instead, it is always installed, but it provides its functionality according to the state of the *demo* dip-switch in the SM. This state is mapped to the state of the Demo aspect, which could be *enabled* or *disabled*[1]. When demo mode is enabled, the corresponding advice replaces the outcome for `ReelsManager.spin()` method. This is similar to the PHANtom implementation where the `OutcomeGenerator` return value is changed.

Besides forcing the prize to be awarded, it is necessary to apply the conflict interaction resolution strategy explained in Sect. 5.2.1. Therefore the `Metering`, Communication Protocols, `GameRecall` and Program Resumption aspects behavior need to be altered. Part of the functionality of these aspects needs to be skipped while playing in *demo* mode. In this case, it is not possible to uninstall them, as AspectJ does not allow for dynamic (un)deployment of aspects. As a workaround, the `Demo` aspect cuts across all the advices of the `ProgramResumption`, `GameRecall` and `Metering` aspects, so that `Demo` can decide to skip or proceed them. Conflicting aspectual behavior can be then avoided. Note that in this case it was decided to avoid the execution of the mentioned aspects, while in the PHANtom version of conflict resolution, for some cases, the approach used consisted of replacing some collaborators by mock objects, for instance, the Meters object was replaced. This is shown in code Listing 6.9.

Listing 6.9: Conflict resolution strategy for Demo, in AspectJ implementation.

```
// Definition of point-cuts in aspects in conflict with demo mode.
 pointcut programResumptionAspectsMethods() :
   execution( * concerns.programResumption.ProgramResumption.*(..)
  );

 pointcut metersAspectsMethods() :
   execution( * concerns.meters.Metering.*(..) );

 pointcut gameRecallAspectsMethods() :
   execution( * concerns.gameRecall.GameRecallAspect.*(..) );

 Object around() :
   programResumptionAspectsMethods()
   || metersAspectsMethods()
   || gameRecallAspectsMethods()
   {
   if(!_on)
     return proceed();
   else
```

[1]Alternatively this could be implemented using the conditional `if` pointcut constructor [56]

```
        return null;
    }
```

The difference between PHANtom and AspectJ based implementations for *conflict* are inherent to the dynamic vs. static features of each language. For this kind of conflict, it is necessary to unweave, or remove some aspectual functionality. If the implementation aspect language or platform does not allow for this, it is necessary to implement a workaround that *simulates* this "unweaving" feature. The Demo aspect captures conflicting aspects advices, so that their behavior can be skipped when Demo mode is enabled.

The result of this limitation in AspectJ is that aspects are installed even though they are not actually active. This is the case of our Demo aspect during a production run and Meters, Communcation Protocols, Program Resumpton and Game Recall during a demo run.

Alternatively, for static aspect oriented languages, another possibility is to have different "builds" of the system. Each build is an instance of the system containing a sound set of aspects. In this way a demo game will not contain, for example, a Program Resumption weaved into the system.

In our experience, certification laboratories requiere the sources and instructions for the build, both in *demo* and *production* modes.

In AspectJ, aspects can be weaved in load time. In this case, a configuration file defines which aspects are weaved before the classes are loaded into the system. This allows to building a *Demo configuration*, which could exclude conflicting aspects, or a production build without Demo functionality.

Note: the last two alternatives require restarting the system in order to run in *demo mode*. These alternative also may seem *fishy* from the certifaction laboratory point of view.

6.4 On the Generic and Explicit Interaction Support

From our implementation experience, we observe that interaction implementation is tangled with the functionality of our crosscutting concerns. For example, consider the case of the Demo aspect. It on the one hand, adds its core functionality (change the outcome of the SM), but on the other hand, it is also responsible for avoiding the execution of other aspects, such as G2S, program resumption or Game Recall. In fact, there is no clear way to distinguish the core functionality of the Demo aspect from interaction implementation. That is to say, interaction implementation is quite implicit. It can be argued that implementing the conflict resolution behavior in a separate aspect is a trivial improvement, but it is necessary to consider that adding conflict resolution in this way adds an extra *dependency* as our Demo aspect will depend on the existence of the conflict resolution aspect.

Besides the maintainability and evolvability problems this could pose, implicit information regarding interactions hinder other tasks such as testing. For example, consider the case of the dependency of G2S on Meters. If the `Metering` aspect is not correctly deployed, the G2S aspect will offer inconsistent data to the server. At the testing phase, testers will connect this SM with a server, play and then observe that the SM keeps reporting wrong data.

This problem goes back to the programmers in the form of: "the SM reports incorrect data" bug report. At this point, some possible sources of this problem are:

1. The Metering aspect is not capturing (all) the relevant events (such as *play*). This happens if it was not deployed, among other possible causes.
2. The G2S protocol for some reason reports incorrect meters, for instance, it takes the values from the wrong meters.
3. There is a bug in serialization so that meter values arrived corrupted.

If the dependency is somehow made explicit in the code, the system would be able to give a warning announcing that the Metering aspect could not be deployed. The system then is able to stop the execution or continue but without G2S support. In this way, a lot of time is saved on testing and debugging. The same idea can be applied to semantic conflicts. If they are explicitily declared the weaver (at compile or run tinme) can raise an error indicating that incompatible behavior is being composed into the system.

These two implementations of interactions presents another drawback: each interaction resolution strategy has been implemented in an *ad hoc* manner. This lack of generality makes it harder to reuse previously applied solutions. We consider this not a major problem, as more interactions implementations need to be studied before a useful generic implementation can be written.

To summarize, the implementations lacks explicit information regarding interactions, it is not possible for the system to check if the system is consistent or not.

6.5 Discussion: PHANtom VS. AspectJ Implementation Results

In this section we summarize the significant differences and similarities found while implementing the interactions with the selected programming languages. Table 6.1 presents an excerpt of the comparison between both implementations.

Table 6.1: Comparison of interaction implementation done using AspectJ and PHANtom.

Interaction	PHANtom	AspectJ
Dependency	Reference + deploy check	Reference + deploy check
Reinforcement	Ad-hoc logic (+ null object)	Inter-type declaration forces considering reinforcement
Mutex	Dedicated aspect + interception of the JP	Dedicated aspect + interception of the JP
Conflict	Run-time (un)weaving + interception	*Ad-hoc* (de) activation logic

The different interaction types found in the SM software have been implemented successfully, although the degree of enforcement of the interaction, the mechanism used to implement, and the visibility and explicitness of programmer's intention vary.

Dependency The dependency for both cases is implemented as a reference to the meters that is required by the communication protocols in order to read and report data. Both implementations share the need for checking the required aspect is installed. Both languages provide means to check this. PHANtom additionally allows to eventually deploy the required aspect. AspectJ offers support in order to force a compile-time check ensuring the required aspect is woven.

Reinforcement This interaction requires implementing some kind of check. In the case of PHANtom based implementation, this is done using ad-hoc logic for ensuring a decision was made for each error condition. In the case of AspectJ, this ad-hoc logic is replaced by the compiler checks, which ensure that there is a method definition for an abstract inherited one. Hence, a missed error condition can be found at compile time in the AspectJ based implementation. Instead, in PHANtom, the missed error condition will be discovered at run-time, the first time it is raised.

Mutex Both implementations apply an aspect which captures the joinpoints where communication protocol mutual exclusion (at configuration item level) must be enforced. However, there are some minor differences between both implementations. The— AspectJ based implementation defines a hierarchy of aspects providing the mutex behavior. The abstract Mutex aspect must be sub-classed for each different configuration item (or their corresponding command) to be configured. Furthermore, the AspectJ based implementation allows to define different criteria for mutex. The differences mentioned before derive from low level design decisions, and are not related to the features offered by the languages used.

Conflict This interaction presents notable differences, originating in the programming language capabilities. As PHANtom is a dynamic aspect language, it allows for run-time weaving and unweaving of aspects. This feature is used by the conflict resolution strategy whenever a conflicting aspect needs to be disabled. In this case, the aspect is removed from the system and re-installed when the conflict is gone. The static approach of AspectJ prevents us of performing run-time weaving. It is thus necessary to simulate the activation and deactivation of aspects. This can be achieved by intercepting the advices of the conflicting aspects, and avoiding their execution by the use of an `around` advise which does not call `proceed()`. This kind of implementation has some disadvantages, for example:

- It makes debugging more complex, as not all the installed aspects can be considered as "active", since the conflict resolution strategy may avoid the execution of the behavior of the mentioned aspects. Furthermore, aspects (such as Demo) are installed even though they are not necessary most of the time.

- This mechanism creates *extra* coupling between aspects, since the aspect that deactivates others needs, of course, refer to the controlled aspects.

6.6 Interactions Extensions for AspectJ

From the implementation of the interactions, we observed that new interaction features are welcome in AOP languages. The examples in literature and our case demonstrate that interactions may materialize in several forms. As a summary of the lessons learned during the implementation stage, we propose some extensions to AspectJ. These extensions are aimed at making interactions explicit in the code. In addition, the information expressed there can be used by the aspect weaver and/or an execution monitor in order to verify their fulfillment (through run-time checks).

We considered AspectJ as the base for the extensions as it is the most widely accepted AOP language. However, the ideas behind them can be transferred to other languages or tools. We will also consider joinpoint based interactions, as they have been acknowledged by several authors. The implementation of these extensions is considered as part of our future work.

6.6.1 Conflict

At joinpoint level, it can be specified similarly to the `hidden_by` semantics of WEAVR. Listing 6.10 shows our proposed syntax: a new `declare` statements (where hence AspectJ's type-patterns can be used). Note that only two parameters can be given.

Listing 6.10: Joinpoint conflict extension for AspectJ.

```
declare advice_conflict: AspectA, AspectB;
```

In this case AspectA and AspectB can be installed at the same time but, for a given joinpoint it is not possible to have advices of AspectA and AspectB being executed. As the intersection of pointcuts can be calculated at compile time, it is possible to stop the weaving process when a joinpoint conflict is detected.

At aspect level, we learned from PHANtom that it is very useful to feature dynamic aspect deployment. Extending AspectJ with such a feature is desirable, in order to cope with conflicts. To allow this an API or keywords are necessary in order to invoke weaving and unweaving. CeasarJ [4] defines keywords that allow to deploy an instantiated aspect. Note that in this case instantiation does not mean aspect activation but requires support for instantiation. In AspectJ it is however not possible to manipulate aspect instances, we therefore proposed a minimal deployment API in the aspects, the methods `deploy()` and `undeploy()`. These allow for the installation respectively uninstallation of all the instances. By default all aspect are installed. An example for uninstalling would be calling the static method `Metering.undeploy()`. This call should uninstall all the corresponding instances of `Metering` aspect, irrespective of their declared instantiation policy (`perTarget`, `perThread`, etc).

Additionally, it is possible to instruct the weaver to avoid conflicts by declaring semantic conflicts. Listing 6.11 shows the general format for declaring a semantic conflict, meaning that both aspects cannot be installed at same time (again type-patterns are used and only two parameters can be given).

Listing 6.11: Semantic conflict extension for AspectJ.

```
declare conflict: TypePattern1, TypePattern2;
```

In our case study, a combination of dynamic weaving of aspects, plus the corresponding declaration of semantic conflicts, as shown in Listing 6.12, suffice.

Listing 6.12: Semantic conflict declarations for the SM domain.

```
declare conflict: Demo, Metering;
declare conflict: Demo, ProgramResumption;
declare conflict: Demo, SCP;
declare conflict: Demo, G2s;
```

Based on this declaration, at the time of the installation of Demo, the conflicting aspects can be uninstalled automatically. Conversely, when Demo is uninstalled , the conflicting aspects are installed.

6.6.2 Dependency

At joinpoint level, dependency can be implemented following the semantics of WEAVR's `depends_on` keyword. This means that if AspectA `depends_on` AspectB, every advice execution of AspectA requires that an advice of AspectB be previously executed. We will informally call this *joinpoint dependency*.

Joinpoint dependency is not useful for semantic dependency, as it is our case for Metering and communication protocols aspects. For this case, we need that an aspect requires another aspect to be installed, we informally call this *semantic dependency*. This can be seen as syntactic sugar that hides the execution of the tests performed using `aspectOf()` in AspectJ (see Sect. 6.3.1), or `isInstalled`, for PHANtom (see Sect. 6.2.1). The proposed notation is presented in Listing 6.13 (again type-patterns are used and only two parameters can be given).

Listing 6.13: Dependency extensions for AspectJ

```
//Joinpoint based dependency
declare joinpointdepends: TypePatternA, TypePatternB;
//Semantic dependency
declare depends: TypePatternA, TypePatternB;
```

Unsatisfied dependencies found during weaving stops the process.

In our case, the communication protocol aspects would be declared as in Listing 6.14.

Listing 6.14: Dependencies for SM domain.

```
declare depends:G2S,Metering;
declare depends:SCP,Metering;
```

If the dependencies are unsatisfied, *i.e.,* Metering is not present when G2S and SCP are woven, the weaving process fails.

6.6.3 Reinforcement

Reinforcement is a less frequent interaction in the literature, therefore it is difficult to propose a generic mechanism for supporting it. According to our experience, it can be generically thought as a functionality that needs to be called in case other (optional) functionality is executed [74].

To express this, indicate the optional functionality must be indicated, as well as the new functionality that *should* be called as a consequence. More concretely, we can say that if `joinpoint1` occurs then `joinpoint2` must occur later. If this does not happen, the system behavior is considered inconsistent, and a corresponding error handler block is executed.

Besides the two involved joinpoints, it is necessary to establish the limits of the computation where `joinpoint2` can occur before considering the reinforcement as unsatisfied. We propose to use the control flow to delimit the scope of the reinforcement interaction. The notation for this is shown in Listing 6.15, where the code means that joinpoints matched by `pointcut2` must occur in the control flow of joinpoints matched by `pointcut1`. Note that the reinforcement declaration takes exactly two pointcuts. The concrete application to our case study is presented in Listing 6.16.

Listing 6.15: Reinforcement extension applied to the SM.

```
declare reinforcement: pointcut1, pointcut2
        {  //error handling code for unsatisfied interaction
        }
```

Listing 6.16: Reinforcement extension for AspectJ.

```
declare reinforcement:
             execution(ErrorCondition.execute()),
             execution(ErrorCondition.notifyToSCPServer())
     {  System.exit(1);  }
```

6.6.4 Mutex

For mutex interactions at joinpoint level, we consider the support provided by Reflex as adequate, since it avoids the execution of two mutually exclusive advices.

Listing 6.17: Mutex extension for AspectJ.

```
1  declare joinpointmutex:   TypePattern1, TypePattern2;
```

We propose a notation for mutex in Listing 6.17. Line 2 shows the general form of mutex declaration, AspectJ's type-patterns are used to denote the two mutually exclussive elements. That is, for a given joinpoint, if pointcuts of both the first and the second match, only the advices of one of the two aspects will be executed. As we have seen in Sect. 6.3.3, mutex may consider policies for determining preference. A default one could be: "the first aspect has precedence", or stop weaving with a meaningful error message. In this case, the semantics of this is similar to `hidden_by` keyword of WEAVR.

At a semantic level, as in our case, it is necessary to express that certain mutually exclusive joinpoints cannot occur. The scope of the interaction must be provided. Again we consider control flow as the possible scoping source, which leads us to the code in Listing 6.18, it declares that either the joinpoints captured by `pointcut1` or the joinpoints captured by `pointcut2` can occur, but not both. Note that this can only be detected at runtime, so it is necessary to declare an error handling block associated with the interaction.

Listing 6.18: Scoped mutex extension for AspectJ.

```
declare mutex: pointcut1, pointcut2, scopingPointcut
  { //error handler block for unsatisfied mutual exclusion
  }
```

In our case, two message sends of the same message of the same class must be mutually exclusive, for example the `execute()` method in the `SetTimeCommand`. What we want to avoid is the call of this method (during a run of the SM) from different communication protocols. So we need to specify the control flow of the main method as part of our mutex declaration, as shown below in Listing 6.19.

Listing 6.19: Scoped mutex extension for AspectJ.

```
declare mutex:
     call(SetTime.execute()) && cflow(call(SCP.process(..))),
     call(SetTime.execute()) && cflow(call(G2S.process(..))),
   in:  execution (SlotMachine.main(..))
     {System.exit(1);}
```

6.7 Conclusions

In this chapter the implementation of the interactions has been presented using dynamic and static aspect languages. These two implementations highlighted differences derived from the capabilities of the two aspect languages. We observed that compiler checks of AspectJ helps in the implementation of the reinforcement, while at the same time it falls short for conflicts where a more dynamic approach is needed. Due to the limitations of AspectJ, it was necessary to simulate aspect deactivation for conflict resolution. This results in extra code and logic that hinders the maintainability of the AspectJ implementation of interactions.

On the other hand, the dynamic features of PHANtom allowed for run-time weaving and unweaving of conflicting aspects, making the implementation more clear and maintainable. This flexibility comes at the cost of having almost no checks at compile time, so reinforcement needs to be implemented as an ad-hoc check that ensures the interaction is satisfied. Once again, this adds extra code that needs to be maintained.

Finally, we presented a set of extensions to AspectJ that to express the the semantic interactions we face in this work and also the joinpoint based interactions reported in the literature. The reimplementation of the interactions using this extension is considered future work.

Chapter 7

Conclusions and Future Work

Aspect Oriented Software Development is arguably not yet a mainstream practice. Its applicability to complex systems still needs investigation so that remaining impediments for its adoption can be removed. One of these impediments is the inherent difficulty in understanding the behavior of the system that is built. This understanding is hindered by the fact that aspects can interact in unexpected ways. Such interactions must however not always be avoided. On the contrary, there are cases where the interactions of crosscutting concerns must be controlled and implemented correctly in order to obtain the desired behavior. Aspect interactions need to be correctly handled and in order to accomplish this objective adequate tools, language constructs and modeling methodologies must be available. Developing them only can be done when having a deep knowledge of the nature of aspect interactions .

The main contribution of this thesis is the identification, modeling and implementation of interactions. During the development cycle of an industrial and non-trivial piece of software. In each phase we evaluated the existing support for interaction of two emblematic aspect oriented approaches. For some of the evaluated approaches, extensions for the explicit support of interactions have been presented.

Our case: software for slot machines (SM), is complex and presents many interacting crosscutting concerns. One of the reasons for this is the fact that the requirements come from different sources. They include government regulations, technical specifications, and standards. As the authors of these requirements have fundamentally different backgrounds and interests, the concerns extracted through requirements analysis exhibit numerous interactions. We classified them according to Sanen's taxonomy [74], in conflict, mutex, dependency and reinforcement. Notably, the SM presents concrete examples for each of these interactions.

In order to develop an aspect oriented version of the SM software, it was necessary to express the interactions in the requirements models, in the design models and during implementation. We evaluated the capability of existing approaches to carry out this task.

In the requirements phase, discussed in Chapter 4, Theme/Doc [9] and

MDSOCRE [60] were used to model requirements, concerns and interactions. We found different limitations such as lack of granularity in Theme/Doc, and absence of interaction expressing features for both Theme/Doc and MDSOCRE. For both approaches we proposed extensions. These included a new graphical notation for Theme/Doc allowing to improve granularity and expressiveness, and new XML constructs in order to improve legibility of MDSOCRE regarding interactions. These extensions have been experimentally demonstrated to be useful for expressing the interactions in this domain.

With this information at hand, in Chapter 5 we studied how to express at the design level how our aspect oriented SM software provides the behavior. Once again, it was necessary to express several design decisions regarding interactions. We therefore evaluated two AOM approaches in order to assess their capabilities: the continuation of Theme/Doc, called Theme/UML, and WEAVR, which has an industrial background and support for interactions. We however found that the interaction support of Theme/UML is poor, and only oriented to fix low level conflicts, such as name clashes, conflicting visibility or cardinality. WEAVR's interactions support is far more advanced than Theme/UML's, but it is intended to solve joinpoint level dependencies or conflicts and therefore is also lacking. Overall, we were able to define the desired behavior at design level but at the cost of making important information regarding the interactions implicit. That is, it is possible to describe the behavior for each interaction, but there is no way to explicitly document the relationship between the artifacts that implement it and the interactions they model.

Once the design decisions regarding interactions had been (roughly) modeled, in Chapter 6 two AOP languages were evaluated for the implementation. AspectJ, arguably the most popular AOP language, was elected as the representative of static aspect oriented languages. On the dynamic languages side, PHANtom was selected to implement the same set of interactions. Considering the four types of interactions, only reinforcement benefited from compiler checks, since it was implemented using intertype declarations. Dependency and mutex were similar, both requiring ad-hoc logic. Conflict benefited from dynamic weaving in PHANtom, and required intrumentation of conflicting aspects in AspectJ. Based on this implementation experience, we proposed language extensions for AspectJ. The extensions deal with joinpoint based and semantic interactions, and may be implemented as a combination of checks at compile time and run time.

After reviewing the research work on aspect interactions and comparing it with the interactions present in the slot machine domain it is not surprising that aspect interaction support falls short in most cases, especially for semantic interactions. In interactions research, we observed that semantic interactions have been neglected in favor of joinpoint based interactions. Even though joinpoint interactions are supported in (some of) the approaches we evaluated, there is no report of a complex industrial system development that validates their suitability. In each phase of the development process of the SM software we observed that aspect orientation lacks maturity regarding interaction support. In spite of the absence of built-in interaction support, in most cases a workaround could be found, but this always made the interactions implicit.

We believe that some interactions can be recurrent in their form for certain domains. At least for the SM, we found more examples for each interaction type, and for each type, all the instances behave in a similar way. For example, all the configuration commands present a mutex interaction that may be resolved in the same way.

Comparing the surveyed interaction examples with those from the SM domain demonstrates that in general interactions take different forms. Hence , the form of interactions is too specific for comprehensive support. Nonetheless, some kind of basic support should be added to aspect oriented approaches for all stages of software development. In this direction, we presented explicit support for AORE approaches in this work. In the design phase interaction behavior was expressed, but explicit support should be developed such that the interactions can be made explicit in the design. General purpose aspect oriented languages also need support for interactions, therefore we presented some possible extensions to AspectJ that need further study.

As a final reflection, it can be concluded that the earlier in the development development stage, the more generic interactions are. Therefore, the more easier is to provide support for them. In later development stages, as the software turns concrete, interactions forms gets more specific, making their support more difficult.

7.1 Towards a Full Aspect Oriented Development Cycle

AOP ideas have infused into many stages in the development process. Besides a good selection of aspect (programming) languages, there is a plethora of aspect oriented requirement analysis and modeling approaches, as reported in several surveys [20, 75, 94]. However, it is clear to us that aspect orientation is not yet industry strength. From our experience we report which needs are covered and which remain unsatisfied.

7.1.1 AORE

Support for crosscutting requirements is the defining feature of AORE approaches. However the granularity used to describe such relationships varies. Somewhat surprisingly, some approaches, such as Theme/Doc [9], do not provide clear information regarding which requirements cut across others.

AORE approaches generally neglect interactions with the exception of "conflict". As the semantics of "conflicts" vary among the different approaches, the support provided also differs. It ranges from including detection of conflicts, contribution analysis and decision making (in goal oriented approaches) to avoiding conflicts. Therefore most of the approaches lack elements for modeling them.

We also noted a deficiency regarding scalability. Many approaches recognize the need for tools in order to scale the number and complexity of requirement models. Yet at the same time, we found that the few tools proposed are unmaintained.

7.1.2 AOM

In our opinion, AOM tools, apart from a few rare exceptions, are clearly behind AORE and AOP in terms of maturity.

Interactions have been considered mostly at a syntactic (joinpoint) level. New mechanisms are needed in order to cope explicitly with semantic interactions. Otherwise this information may be lost, even though it is critical for the correct behavior of the system.

The differences in the notation for some common concepts of aspect orientation is, in our opinion, a distracting factor. Thus, unifying the modeling languages is needed, at least for basic AO concepts such us crosscuts, advice, aspects, etc.

AOM shares the same scalability problem of AORE and lack of supporting tools. Tools for AOM can be classified regarding their objective: modeling, weaving and code generation. The absence of tool support for modeling and the resulting lack of scalibility can be alleviated by the use of abstractions that allow to apply the same solution to similar scenarios. This may reduce the amount of design documents to be maintained. However, tool support is needed for automatic weaving of models. We consider the modeling and weaving tools as being critical for the success of AOM. In contrast, code generation, while desirable, is not essential for the future of AOM.

7.1.3 AOP

Developers dislike when their code is altered by other developers or programs, especially if these changes cannot be properly visualized. The nature of aspects is to typically alter the behavior of several modules. This is confusing for developers and may be one of the reasons for them being reluctant to adopt AOP languages. In this regard, tool support to adequately visualize where the aspects crosscut is needed. The AspectJ development tools provide visualization and navigation tools. However, the visualization presents deficiencies such as poor scalability and complexity problems. More advanced tools such as AspectMaps [2] overcome or alleviate these problems. AspectMaps provides structural zoom and other facilities that improve the understanding of the system. However, it is a new tool and time is needed to evaluate its applicability to industrial problems.

Aspect interactions complicate the scenario, as they can occur in unanticipated ways. Therefore, having support for interaction detection is important, as a result there are many languages that claim to provide joinpoint level interaction detection. Moreover, having tools/constructs that allow the programmer to clearly and explicitly define the behavior of the system when interactions occur is key.

The current support for implementing aspect interactions in AOP languages is poor, however it is possible to implement interaction using ad-hoc logic. This usually makes implicit the implementation of such interactions, with the resulting maintainability problems.

7.1.4 From the Lab to the Industry

AOSD approaches need to accomplish the migration from the academy to the industry. While reviewing the existing literature we found few industrial applications of them. This results in immature approaches that work mostly for limited and well selected case studies.

In the academia, initial case studies may be carefully selected to match the capabilities of the tool to be presented. Many implementations we studied are proofs of concept that did not need to deal with certain issues. Finally, as we observed in some AORE approaches regarding interactions, it is possible to *skip* certain problems instead of modeling them: consider for example the conflict management of Theme/UML, or the conflict removal process of MDSOCRE.

The next step is therefore to try the existing approaches in the "real world". Industrial and commercial systems present new challenges for these approaches. Requirements come from different competing stakeholders. Trade-offs are the rule. The low time to market highlights the need for lightweight methodologies, as there is not much time for modeling and documentation. Changes in the requirements and the need for new versions of the software containing those changes call for traceability mechanisms.

The evolution of the AOSD approaches needed to support these scenarios takes time and effort. Any tool presents a lot of inconveniences in its initial versions. User communities are needed and continuous evolution and maintenance effort is critical to reach a minimal industrial maturity level. A good example is AspectJ, which has been available during more than a decade and holds the throne of the most influential aspect oriented language.

7.1.5 The Missing Link: Traceability

Traceability from requirements to the implementation and back again is vital to many software engineering activities [81] and still remains as an open problem in this field [49].

We found some support for traceability during this experience. In the case of Theme/Doc and Theme/UML they are supposed to seamlessly integrate. A theme in Theme/Doc maps directly to a theme in Theme/UML. The Theme documentation proposes to propagate changes made at design time back to the requirements in order to keep models consistent. However, there are no explicit process or tools for doing so. From design to implementation, Theme proposes some rules to map Theme/UML designs to code in an aspect oriented programming language. Following this rule implies the use of abstract aspects named as themes, and concrete aspects defining their bindings as concrete pointcuts. Somewhat surprisingly, MDSOCRE does not consider the traceability problem.

In the case of WEAVR there is no AORE approach associated. Concerns are modeled as state machines. Crosscutting concern state machines are woven into other machines based. We believe that any AORE approach can be used with WEAVR, as long as consistency between concerns in the requirements analysis phase and in the design phase is maintained. Some special attention would be needed in order to highlight those requirements that define states, state changes and weaving points. As WEAVR includes code generation, traceability from design to implementation is direct. However, it is not specified whether the generated code can be modified and those modifications preserved in later builds.

7.2 Contributions and Related Publications

We summarize below the contributions of this work and the associated publications:

- An aspect oriented development cycle of the SM software has been carried out, considering interactions from the onset. To the best of our knowledge, this the first report of a complete development cycle of such complex software. This resulted in the following three contributions.

- Two AORE approaches have been assessed and extended. These extensions have been evaluated experimentally in order to judge their applicability. This work is published in [97, 98, 99]

- AOM modeling approaches that claim to support interactions, even in industrial settings, have been tested against the interactions in the SM domain. They proved to fall short in most of the cases. This work is published in [36].

- Static and dynamic AOP languages have been used to implement ad-hoc solutions for these interactions. The results show that their dynamic or static nature has a strong impact on the implementation of some interactions. We also proposed a set of extensions for an aspect oriented language that cover all the interactions found plus the joinpoint interactions reported in the literature. Even though the results of this stage are not yet implemented, some of them are inspired on our work published in [100].

- An industrial case study revealing a complex domain has been studied in depth, resulting in a sizable amount of crosscutting concerns being found. Many of them are functional concerns.

- Genuine aspect interaction examples have been found and documented. We believe the interactions found serve as a challenging test bed for upcoming AOSD approaches.

7.3 Future work

7.3.1 General

Interactions in other Domains

The slot machines domain presented a good amount of functional and non-functional concerns. There were also many examples of the different kinds of interactions. To the best of our knowledge, there is no other report in the literature with such a high amount of interactions. This fact makes us wonder if other domains exhibit as many concerns and interactions as SM. There are two possible results we would like to confirm:

- There are more domains, which can indicate that existing studies have overlooked interactions, possibly due to lack of deep domain knowledge.

- Other domains present a lower number of crosscutting concerns and interactions. On the positive side, this means that the SM is an exceptionally good case study to test the power of different approaches. On the negative side, this means that the obtained results maybe not easily applied to other domains.

Traceability

Traceability of interactions needs to be studied. We know that requirement sources change at different speeds. A change in some requirement of a crosscutting concern may impact in several aspects if it is part of an interaction. For example, a change in the requirements for meters impacts not only in the Meters aspect, but also on the communication protocols, whose code need to be updated in order to keep reported data consistent. Therefore, support for practical traceability from requirements to interactions in design and implementation is needed, in order to make a consistent change set.

7.3.2 Requirements Analysis

A first avenue of future work would be a study of the impact of our extensions to evolution of requirements. Second, the size of our requirements base and the fact that these come from different sources with their own formats, makes it desirable to automatize how the *modeled* version of the requirements is obtained and evolved from one version to another. We believe that extra information represented by the interactions can help to better track the impact of changes, but this needs to be corroborated.

Although we were able to improve the expressiveness of Theme/Doc, a detailed experiment could be useful to compare it with the results obtained in Sect. 4.4.1 for MDSOCRE-I. Also, the deployment of our extensions in the industry will help to analyze other issues such as scalability or requirements evolution.

7.3.3 Design

The key question for future work is how we would be able to satisfactorily express the interactions in our design. The most straightforward solution would be to extend one of the above methodologies such that it includes the support we are lacking. We consider this therefore as the main avenue for future work.

From the evaluation performed on WEAVR and Theme/UML we envision useful extensions for those approaches. The lack of explicit support for mutex, reinforcement and dependency in Theme/UML could be solved by extending the notation. Conflict could benefit of such extensions, but some of the resolution strategies explained in Sect. 5.2.2, need further additions. In this case, Theme/UML requires to be extended in order to model the weaving process, allowing to express directives for aspect deployment.

In the case of WEAVR, conflict and dependency support is only at joinpoint level, thus an extension to aspect level could help to make our interactions more explicit. Mutex is not supported, and the ad-hoc solution found (composition of the mutual exclusive state machines) clearly poses a scalability problem.

We believe that recursive composition of WEAVR's state machines may overcome this problem. For reinforcement, the state machine execution engine of WEAVR might be useful for running simulations, and checking if the optional behavior gets called. However, this needs to be evaluated. Separately, it is necessary to make reinforcement explicit, so that the engineer can easily identify when the simulation needs to be run again.

Part of our future work is also the application of a third AOM approach, called RAM [55], which claims to support interactions and conflicts.

7.3.4 Implementation

Configuration vs Sanity Check

From Chapter 6 we conclude that aspects implementing crosscutting concerns and interaction resolution strategies are overloaded with responsibilities. Decoupling (part of) the interaction resolution implementation helps to relieve them of extrinsic responsibilities, improving maintainability.

From our implementations, we observed that part of the resolution of interactions can be implemented by controlling the *deployment configuration* of our aspectual system. By *deployment configuration* we refer to the selection of the included aspects in a given run or deployment.

Specific configurations should be developed for different deployments. In our experience, chances are that the resulting deployment is not fully consistent due to unsatisfied interactions. Therefore a diagnostic mechanism for checking interaction satisfaction is also needed. This mechanism must check that interactions are satisfied, if it is not the case the SM should not run, indicating the error.

Interaction relationships can be expressed in the form of constraints. If the language provides facilities for metadata, constraints can be placed in the source code. Adding metadata to code, in order to express and check constraints or document the code semantics is not a new practice [13, 54]. For example, in [100] we have used annotations to denote the role of aspects and advices in order to manage their activation using a rule engine. Interaction constraints should be the result of the information collected during requirement analysis and design phases. The whole set of constraints for the selected aspects in a given deployment must be checked on start-up. If any constraint fails, the system can be considered to be in an inconsistent state, therefore, it must be shutdown indicating the reason in a error message. This is what we call a *sanity check*. The purpose of this mechanism is to detect unsatisfied interactions in the deployed system.

In our proposal, metadata is used to check the satisfaction of the aspect interaction constraints. The decision of what to do when interactions are unsatisfied is left to the programmer, but some default actions, such as shutdown the system, can be defined.

Separation of Concerns for Interactions Implementation

Several implementations for the different interactions have been discussed and presented. A future work is to analyze the feasibility of generalizing them to find ways of reusing the knowledge acquired and the behavior defined for them.

Before generalizing them, it is however necessary to decouple interaction resolution strategies which are tangled with the aspects' core behavior. Consider the `Demo` aspect, whose core behavior consists of offering the player (certifier in this case) to select a prize, and forcing the outcome for matching the selected prize. Besides this core functionality, our Demo implementation copes with extra responsibilities such as: switching off (un-installing) certain conflicting aspects or replacing behavior in the communication protocols to prevent reporting of demo plays. These extra responsibilities clearly hinder the objective of providing a clear and as modular as possible separation of concerns. Furthermore, if the interaction resolution strategy implementation is separated from the interacting aspects, it is possible to find common interaction resolution behavior in order to generalize it and finally try to reuse it.

In the previous section, we discussed how to decouple part of the interaction resolution strategies by using deployment configurations and the *sanity check* to ensure that the included aspects and their interactions are consistent. Unfortunately, this does not allow for a complete interaction resolution implementation. Some parts of the interaction resolution strategies still need to be implemented programmatically. In these cases, we consider it important to keep them as separated objects or aspects (according to the needs) so that they can be studied, when possible, in isolation.

Implementation of AspectJ extensions for Interactions

As a final avenue for future work we considered the implementation of the proposed extensions to AspectJ in order to support interactions. For many of them it is possible to develop a proof of concept by using code transformation. In this case, we need to develop a parser that takes the extended syntax and converts it into regular AspectJ code that implement the interactions semantic. We consider this approach feasible as it does not involves altering the (complex) AspectJ compiler. Another possible implementation alternative is to use the *abc compiler* [6], which is an open implementation intended to test AspectJ extensions, as it is our case.

Bibliography

[1] Recommended practice for software requirements specifications. *IEEE Std 830-1998*, 1998.

[2] *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*. IEEE Computer Society, 2011.

[3] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting dependences and interactions in feature-oriented design. In *ISSRE*, pages 161–170. IEEE Computer Society, 2010.

[4] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *T. Aspect-Oriented Software Development I*, pages 135–173, 2006.

[5] Gaming Standards Association. Gaming standards association home page, January 2013.

[6] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.

[7] Shubhanan Bakre and Tzilla Elrad. Scenario based resolution of aspect interactions with aspect interaction charts. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, AOM '07, pages 1–6, New York, NY, USA, 2007. ACM.

[8] Elisa Baniassad and Siobhan Clarke. Finding aspects in requirements with Theme/Doc. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development*, March 2004.

[9] Elisa Baniassad and Siobhan Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Elisa Baniassad, Paul C. Clements, Joao Araujo, Ana Moreira, Awais Rashid, and Bedir Tekinerdogan. Discovering early aspects. *IEEE Softw.*, 23(1):61–70, 2006.

[11] David Bar-On and Shmuel Tyszberowicz. Derived requirements generation: The DRAS methodology. *Software Science, Technology and Engineering, IEEE International Conference on*, 0:116–126, 2007.

[12] L. Bergmans. The Composition Filters Object Model. Technical report, Dept. of Computer Science, University of Twente, 1994.

[13] Lodewijk M. J. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[14] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[15] Isabel Brito and Ana Moreira. Integrating the NFR framework in a RE model. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 3rd International Conference on Aspect-Oriented Software Development*, 2004.

[16] Isabel Sofia Brito, Filipe Vieira, Ana Moreira, and Rita Almeida Ribeiro. Handling conflicts in aspectual requirements compositions. *Transactions in Aspect-Oriented Software Development*, 3:144–166, 2007.

[17] Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, and Arend Rensink. Editorial for special section on dependencies and interactions with aspects. 5490:133–134, 2009.

[18] Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, and Arend Rensink. Editorial for special section on dependencies and interactions with aspects. *T. Aspect-Oriented Software Development*, 5:133–134, 2009.

[19] Ruzanna Chitchyan, Awais Rashid, Paul Rayson, and Robert Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48, New York, NY, USA, 2007. ACM.

[20] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke, and Andrew Jackson. Survey of analysis and design approaches. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, University of Lancaster, 2005.

[21] Ruzanna Chitchyan, Awais Rashid, and Peter Sawyer. Comparing requirement engineering approaches for handling crosscutting concerns. In João Araújo, Amador Durán Toro, and João Falcão e Cunha, editors, *8th Workshop on Requirements Engineering held at CAiSE'05*, pages 1–12, 2005.

[22] Lawrence Chung and Brian A. Nixon. Dealing with non-functional requirements: three experimental studies of a process-oriented approach. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 25–37, New York, NY, USA, 1995. ACM.

[23] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. Springer, 1999.

[24] Lawrence Chung and Julio Cesar Prado Leite. Conceptual modeling: Foundations and applications. chapter On Non-Functional Requirements in Software Engineering, pages 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[25] Selim Ciraci, Wilke Havinga, Mehmet Aksit, Christoph Bockisch, and Pim van den Broek. A graph-based aspect interference detection approach for uml-based aspect-oriented models. *T. Aspect-Oriented Software Development*, 7:321–374, 2010.

[26] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design. The Theme Approach.* Object Technology Series. Addison-Wesley, Boston, USA, 2005.

[27] Aspect Oriented Modeling Community. Aspect oriented modeling workshop series home page, January 2013.

[28] Thomas Cottenier, Aswin V. Berg, and Tzilla Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *in Proceedings of the International Conference on AspectOriented Software Development, Industry Track*, 2006.

[29] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola weavr: Aspect and model-driven engineering. *Journal of Object Technology*, 6(7):51–88, 2007.

[30] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[31] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 141–150, New York, NY, USA, 2004. ACM.

[32] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. pages 173–188. Springer-Verlag, 2002.

[33] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October 2001.

[34] Johan Fabry and Daniel Galdames. Phantom: a modern aspect language for pharo smalltalk. *Software: Practice and Experience*, pages n/a–n/a, 2012.

[35] Johan Fabry, Éric Tanter, and Theo D'Hondt. Relax: implementing kala over the reflex aop kernel. In *Proceedings of the 2nd workshop on Domain specific aspect languages*, DSAL '07, New York, NY, USA, 2007. ACM.

[36] Johan Fabry, Arturo Zambrano, and Silvia Gordillo. Expressing aspectual interactions in design: Experiences in the slot machine domain. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2011.

[37] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[38] David Flanagan and Yukihiro Matsumoto. *The ruby programming language.* O'Reilly, first edition, 2008.

[39] B. D. Fraine, P. D. Quiroga, and V. Jonckers. Management of aspect interactions using statically-verified control-flow relations. In *Proceedings of the 3rd International Workshop on Aspects, Dependencies and Interactions*, 2008.

[40] Gaming Laboratories International. *Gaming Devices in Casinos*, 2007. Available at: http://www.gaminglabs.com/.

[41] Gaming Standard Association. *Game to Server (G2S) Protocol Specification*, 2008. Available at: http://www.gamingstandards.com/.

[42] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Prentice Hall, June 2005.

[43] John C. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 84–91, Washington, DC, USA, 1999. IEEE Computer Society.

[44] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *5th International Conference on Aspect-Oriented Software Development, AOSD*, pages 214 – 225, January 2006.

[45] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[46] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, NODe '02, pages 216–232, London, UK, UK, 2003. Springer-Verlag.

[47] Z. 100: ITU. Specification and description language (sdl). In *International Telecommunication Union.* 2000.

[48] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2004.

[49] A. Kannenberg and H. Saiedian. Why software requirements traceability remains a challenge. *The Journal of Defense Software Engineering*, 22(7):14–19, 2009.

[50] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In Curtis Clifton, editor, *FOAL*, pages 29–38. ACM, 2008.

[51] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, number 4067 in LNCS, pages 501–525, 2006.

[52] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[53] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[54] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[55] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 87–98, New York, NY, USA, 2009. ACM.

[56] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[57] Jia Liu, Don S. Batory, and Srinivas Nedunuri. Modeling interactions in feature oriented software designs. In *Feature interactions in telecommunications and software systems VIII*, pages 178–197, 2005.

[58] Antoine Marot. *Preserving the Separation of Concerns while Composing Aspects with Reflective AOP.* Phd thesis, Université Libre De Bruxelles, October 2011.

[59] Katharina Mehner, Mattia Monga, and Gabriele Taentzer. Interaction analysis in aspect-oriented models. In *RE*, pages 66–75. IEEE Computer Society, 2006.

[60] A. Moreira, A. Rashid, and J. Araujo. Multi-dimensional separation of concerns in requirements engineering. In *Proc. 13th IEEE International Conference on Requirements Engineering*, pages 285–296, 29 Aug.–2 Sept. 2005.

[61] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. Inquiring the usage of aspect-oriented programming: an empirical study. In *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, Canada, 2009.

[62] Gunter Mussbacher, Jon Whittle, and Daniel Amyot. Semantic-based interaction detection in aspect-oriented scenarios. In *RE*, pages 203–212. IEEE Computer Society, 2009.

[63] Nevada Gaming Commission. *Technical Standards For Gaming Devices And On-Line Slot Systems*, 2008. Available at: http://gaming.nv.gov/stats_regs.htm.

[64] Nan Niu, Yijun Yu, Bruno González-Baixauli, Neil A. Ernst, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. Aspects across software life cycle: A goal-driven approach. *T. Aspect-Oriented Software Development VI*, 6:83–110, 2009.

[65] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[66] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A uml notation for aspect-oriented software design. In *IN WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML (AOSD-2002*, 2002.

[67] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27:58–93, January 2001.

[68] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, and W. Joosen. Aspect-oriented software development in practice: Tales from aosd-europe. *Computer*, 43(2):19 –26, feb. 2010.

[69] Awais Rashid and Ana Moreira. Domain models are NOT aspect free. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MODELS06)*, volume 4199 of *Lecture Notes in Computer Science*, pages 155–169. Springer Verlag, October 2006.

[70] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 11–20, New York, NY, USA, 2003. ACM.

[71] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition.* New Riders Games, 2003.

[72] Americo Sampaio, Phil Greenwood, Alessandro F. Garcia, and Awais Rashid. A comparative study of aspect-oriented requirements engineering approaches. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 166–175, Washington, DC, USA, 2007. IEEE Computer Society.

[73] Américo Sampaio, Awais Rashid, Ruzanna Chitchyan, and Paul Rayson. Ea-miner: Towards automation in aspect-oriented requirements engineering. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development III*, volume 4620 of *Lecture Notes in Computer Science*, pages 4–39. Springer Berlin / Heidelberg, 2007.

[74] Frans Sanen, Eddy Truyen, Bart De Win, Wouter Joosen, Neil Loughran, Geoff Coulson, Awais Rashid, Andronikos Nedos, Andrew Jackson, and Siobhan Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[75] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, and M. Wimmer. A survey on aspect-oriented modeling approaches. Technical report, Vienna University of Technology, 2007.

[76] Devon Simmonds, Raghu Reddy, Robert France, Sudipto Ghosh, and Arnor Solberg. An aspect oriented model driven framework. In *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, EDOC '05, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.

[77] Ian Sommerville, Ian Sommerville, Pete Sawyer, and Pete Sawyer. Viewpoints: principles, problems and a practical approach to requirements engineering. *Annals of Software Engineering*, 3:101–130, 1997.

[78] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in aop with aspectc++. In *Proceedings of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fourth SoMeTW05*, pages 33–53, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press.

[79] Spring. Springframework reference manual 3.1. 2011.

[80] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Senthil Karthikeyan Sundaram, Jane Huffman Hayes, Alex Dekhtyar, and Elizabeth Ashlee Holbrook. Assessing traceability of software engineering artifacts. *Requir. Eng.*, 15(3):313–335, 2010.

[82] Éric Tanter. Aspects of composition in the reflex aop kernel. In *Proceedings of the 5th international conference on Software Composition*, SC'06, pages 98–113, Berlin, Heidelberg, 2006. Springer-Verlag.

[83] Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, March 2010. ACM Press. Best Paper Award.

[84] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language aop. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[85] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[86] Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton, and Jr. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[87] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[88] Bruno De Fraine Thomas Cleenewerck, Johan Brichau. Conflict resolution strategies. Technical report, Aspect Lab II, AOSD Europe, 2007.

[89] Claire Tristram. The Technology Review 10: Emerging Technologies that Will Change the World. 1:97–103+, 2001.

[90] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society.

[92] Jon Whittle and João Araújo. Scenario modelling with aspects. *IEE Proceedings - Software*, 151(4):157–172, 2004.

[93] Jon Whittle and Praveen Jayaraman. MATA: A tool for aspect-oriented modeling based on graph transformation. In Holger Giese, editor, *Models in Software Engineering: Workshops and Symposia at MoDELS 2007*, volume 5002 of *Lecture Notes in Computer Science*, pages 16–27. Springer Berlin / Heidelberg, 2008.

[94] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. A survey on uml-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):28:1–28:33, October 2011.

[95] Bobby Woolf. Pattern languages of program design 3. chapter Null object, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[96] Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 38–47, Washington, DC, USA, 2004. IEEE Computer Society.

[97] Arturo Zambrano, Johan Fabry, and Silvia Gordillo. Expressing aspectual interactions in requirements engineering: Experiences, problems and solutions. *Science of Computer Programming*, 78(1):65 – 92, 2012. Special Section: Formal Aspects of Component Software.

[98] Arturo Zambrano, Johan Fabry, and Silvia Gordillo. *Aspect-Oriented Requirements Engineering*, chapter Experience Report: AORE in Slot Machines. Springer Verlag, To appear in August 2013.

[99] Arturo Zambrano, Johan Fabry, Guillermo Jacobson, and Silvia Gordillo. Expressing aspectual interactions in requirements engineering: experiences in the slot machine domain. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 2161–2168. ACM Press, 2010.

[100] Arturo Zambrano, Silvia Gordillo, and Johan Fabry. A fine grained aspect coordination mechanism. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 20(7):1025–1042, November 2010.

Appendix A

Theme/Doc Diagrams

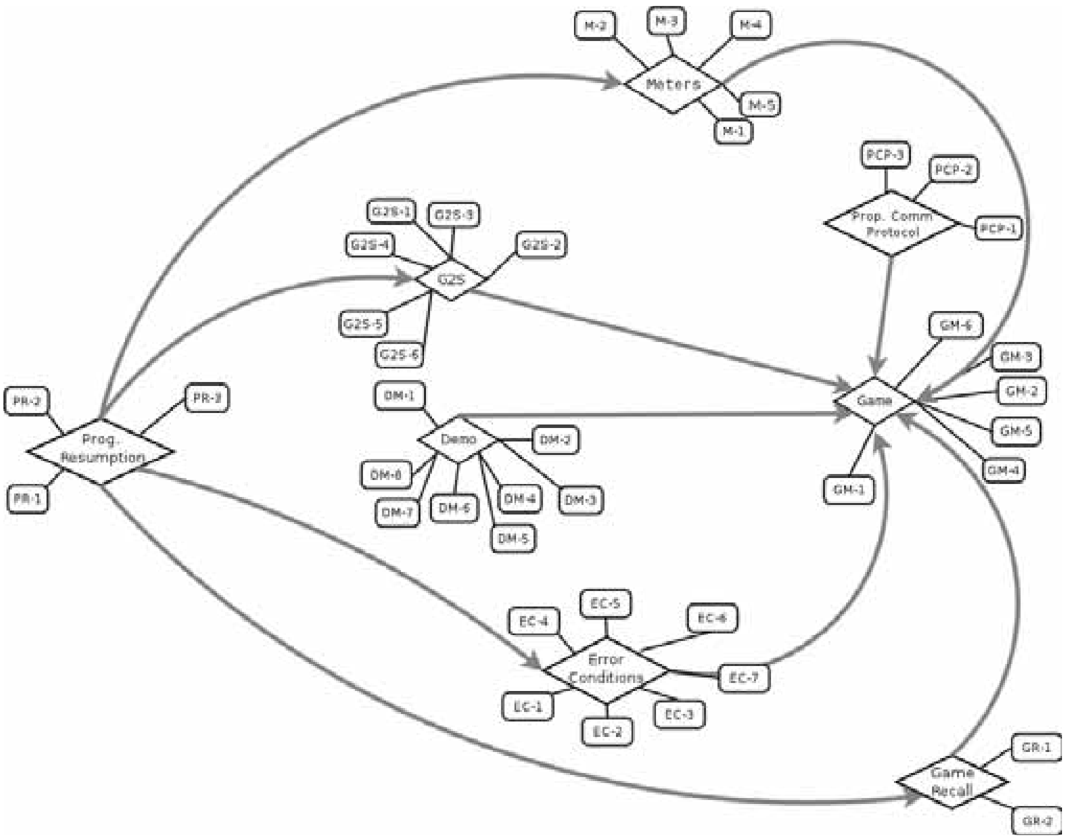


Figure A.1: Theme/Doc approach applied to the selected requirements and concerns

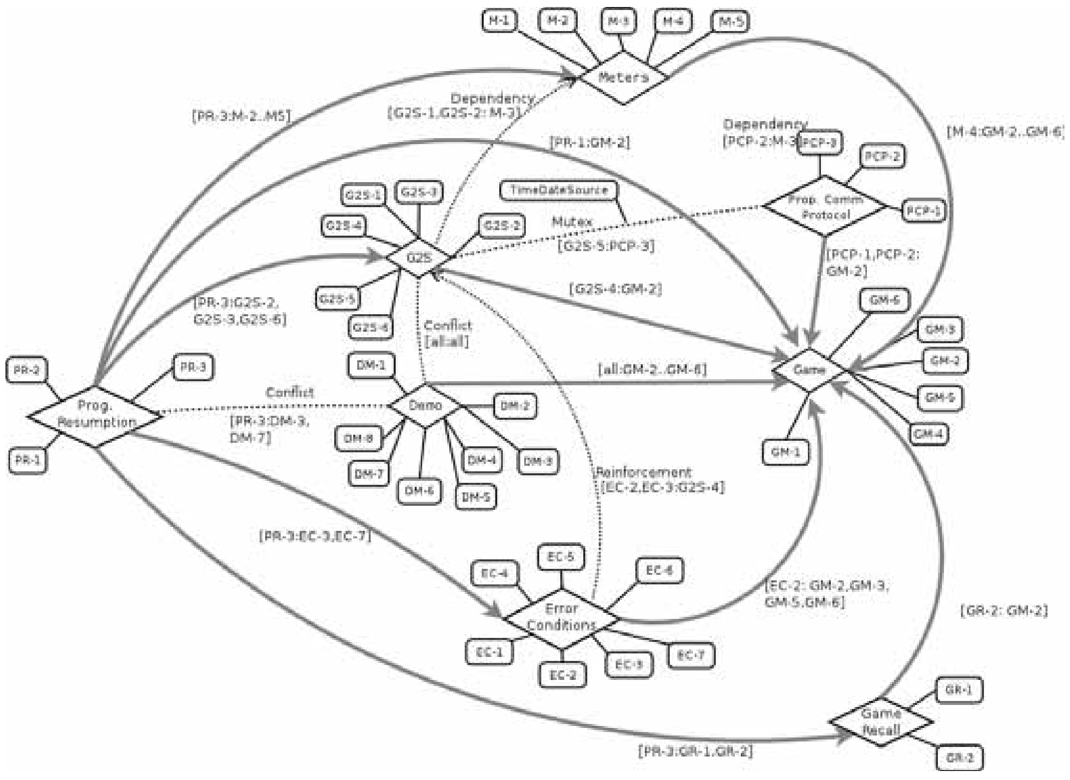


Figure A.2: Extensions to Theme/Doc applied

Appendix B

MDSOCRE Code Listings

Listing B.1: Concerns and crosscutting relationships for selected requirements

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <Concern name="Game">
        <Requirement id="1"> A slot machines have 5 reels.
        </Requirement>
        <Requirement id="2"> Reels spin when play button is pressed.
        </Requirement>
        <Requirement id="3"> Prizes are awarded according to a pay
            table.
        </Requirement>
        <Requirement id="4"> A slot machine has  one or more devices
             for entering money.
        </Requirement>
        <Requirement id="5"> As money is inserted credits are "
            assigned" to the player.
        </Requirement>
        <Requirement id="6"> A slot machine must provide means for
            cashing the credits out.   It could be a ticket printer,
             a coin hopper.
        </Requirement>
    </Concern>

    <Concern name="Meters">
<!-- From GLI 11 -->
        <Requirement id="1"> Credit meter: shall at all times
            indicate all credits or cash available for the player to
             wager or cashout  (GLI 11 4.10.1)
        </Requirement>
        <Requirement id="2"> Credit Meter Incrementing: The value of
             every prize (at the end of a game) shall be added to
            the player's credit meter. The credit meter shall also
            increment with the value of all valid coins, tokens,
            bills, Ticket/Vouchers, coupons or other approved notes
            accepted. (GLI 11 4.10.5)
        </Requirement>
        <Requirement id="3"> Accounting Meters (GLI 11 4.10.9):
            Coin In: a meter that accumulates the total value of all
             wagers [...]. Games-played: accumulates the number of
            games played; since power reset, since door close and
            since game initialisation.
        </Requirement>
        <Requirement id="4"> Meters should be updated upon
            occurrence of any event that must be counted, including:
             play, cashout, bill in, coin in.
        </Requirement>
<!-- From G2S -->
        <Requirement id="5" seeAlso="2,4"> Some G2S meters are:
            gamesSinceInitCn Number of games since initialisation.
            WonCnt: Number of primary games won by the player.
            LostCnt: Number of primary games lost by the player
        </Requirement>
    </Concern>

    <Concern name="Game Recall">
        <Requirement id="1"> Information on at least the last ten
            (10) games is to be always retrievable on the  operation
             of a suitable external key-switch, or another secure
            method that is not available to  the player.
        </Requirement>
        <Requirement id="2"> Last play information shall provide all
             information required to fully reconstruct the last ten
            (10) plays. All values shall be displayed; including the
             initial credits, credits bet, and credits won, payline
            symbol combinations and credits paid whether the outcome
             resulted in a win or loss. This information should
            include the final game outcome, including all player
            choices and  bonus features.
        </Requirement>
    </Concern>

    <Concern name="G2S">
        <Requirement id="1"> The  G2S protocol is designed to
            communicate information between a SM, and one or more
            host systems.
        </Requirement>
        <Requirement id="2"> Meter information can be queried by a
            host in real-time or a host may set a periodic meter
            subscription to cause the EGM to send selected meters at
             predetermined intervals.
        </Requirement>
        <Requirement id="3"> Information provided by the SM is used
            for audit purposes.
        </Requirement>
        <Requirement id="4"> The device can generate an event in an
            unsolicited manner or in response to a host command.
        </Requirement>
        <Requirement id="5"> Current timestamp can be configured by
            the host.
        </Requirement>
  <Requirement id="6"> Command GetGameRecallLog is used by a host to
       request the contents of a transaction log of last playsfrom a
       SM.
  </Requirement>
    </Concern>

    <Concern name="Proprietary Communication Protocol (SCP)">
        <Requirement id="1"> The SCP  communicates a SM with a host
            system.
        </Requirement>
        <Requirement id="2"> It must report all meters of a SM.
        </Requirement>
        <Requirement id="3">Configuration settings such as current
            timestamp are configured from the host.
        </Requirement>
        <Requirement id="4"> If error conditions such as: door open, ticket inserted, paper out, etc. can be detected they shall be informed to the host.
        </Requirement>
    </Concern>

    <Concern name="Program Resumption">
    <!--From Nevada regulation -->
    <Requirement id="1" seeAlso="3"> After a program interruption (e.g., processor reset), the software shall be able to recover to the state it was in immediately prior to the interruption occurring.
         </Requirement>
        <Requirement id="2"> Restoring Power. If a gaming device is powered down while in an error condition, then upon restoring power, the specific error message shall still be displayed and the gaming device shall remain locked-up.
        </Requirement>
     <!-- From G2S  1.16 -->
        <Requirement id="3" seeAlso="1"> A SM must store all meter information in persistent memory.
        </Requirement>
    </Concern>

    <Concern name="Error Conditions and Events">
     <!--From Nevada regulation -->
        <Requirement id="1"> Gaming devices shall be capable of detecting and displaying error conditions and illuminate the tower light for each  or sound an audible alarm.
        </Requirement>
        <Requirement id="2"> Error conditions should cause the gaming device to lock up and require attendant intervention. Error conditions shall be cleared either by an attendant or upon initiation of a new play sequence after the error has cleared except for those deemed as a critical error.
        </Requirement>
      <!-- From GLI  -->
        <Requirement id="3"> Error conditions are: coin jam, reverse coin in, stacker full, bill jam, external doors open.
        </Requirement>
        <Requirement id="4"> Video based games shall display meaningful text as to the error  conditions.
        </Requirement>
        <Requirement id="5"> Error conditions shall be communicated to an on-line monitoring and  control system when this is available.
        </Requirement>
        <!-- From g2s -->
        <Requirement id="6" seeAlso="1">  An event represents an occurrence of an incident detected by a device in an EGM
        </Requirement>
        <Requirement id="7"> Important events must be reported in real-time, including: error conditions, tickets inserted, ticket printed.
        </Requirement>
    </Concern>

    <Concern name="Demo Mode">
      <!-- From Nevada -->
```

```
        <Requirement id="1"> The Slot Machine must permit a a test,
            diagnostic or demo mode, which permits gaming device (e.
            g.,  a hopper test) shall be completed on resumption of
            normal operation.
        </Requirement>
        <Requirement id="2"> If the gaming device is in a test,
            diagnostic or demo mode, any test that incorporates
            credits entering or leaving the gaming device (e.g.,  a
            hopper test) shall be completed on resumption of normal
            operation.
        </Requirement>
        <Requirement id="3"> There shall not be any mode other than
            normal operation (ready for play) that increments any of
             the electronic meters.
        </Requirement>
        <Requirement id="4">  Any credits on the gaming device that
            were added during the test, diagnostic or demo mode
            shall be automatically cleared before the mode is exited
            . </Requirement>
        <!-- From GLI 11 4.17 -->
        <Requirement id="5"> Specific meters are permissible for
            these types of modes provided the meters indicate as
            such
        </Requirement>
        <Requirement id="6"> The main cabinet door of the gaming
            device may automatically place the gaming device in a
            service or test/diagnostic mode. Test/diagnostics mode
            may also be entered, via an appropriate instruction,
            from an attendant during an audit mode access. These
            modes should not be accessible to the player
        </Requirement>
        <Requirement id="7">  When exiting from test-diagnostic mode
            , the game shall return to the original state it was in
            when the test mode was entered
        </Requirement>
        <Requirement id="8"> Test Games. If the device is in a game
            test mode, the machine shall clearly indicate that it is
             in a test mode, not normal play.
        </Requirement>
    </Concern>

  <!--        CROSSCUTTING RELATIONSHIPS      -->

    <Composition>
        <Requirement concern="Demo Mode" id="all">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="all" />
            </Constraint>
            <Constraint action="exclude" operator="on">
                <Requirement concern="Game" id="1" />
                <Requirement concern="Game" id="2" />
            </Constraint>
            <Outcome action="fulfilled"/>
        </Requirement>
    </Composition>


    <Composition>
        <Requirement concern="Game Recall" id="2">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="2" />
            </Constraint>
            <Outcome action="fulfilled"/>
        </Requirement>
    </Composition>


    <Composition>
        <Requirement concern="Meters" id="4">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="2,3,4,5,6" />
            </Constraint>
            <Outcome action="fulfilled"/>
        </Requirement>

    </Composition>


    <Composition>
        <Requirement concern="Program Resumption" id="1">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="2" />
            </Constraint>
            <Outcome action="fulfilled"/>
        </Requirement>
    </Composition>


    <Composition>
        <Requirement concern="G2S" id="4">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="2" />
            </Constraint>
            <Outcome action="fulfilled"/>
        </Requirement>
    </Composition>

    <Composition>
        <Requirement concern="Error Condition " id="3">
            <Constraint action="enforce" operator="on">
                <Requirement concern="Game" id="2,3,5,6" />
            </Constraint>
            <Outcome action="satisfied">
                <Requirement concern=" Error Condition" id="2"/>
                </Outcome>
            </Requirement>
        </Composition>

        <Composition>
            <Requirement concern="Proprietary Communication Protocol
                " id="1,2">
                <Constraint action="enforce" operator="on">
                    <Requirement concern="Game" id="2" />
                </Constraint>
                <Outcome action="fulfilled"/>
            </Requirement>
        </Composition>

</root>
```

Listing B.2: Interactions for the selected requirements expressed using MD-SOCRE notation

```
<Composition>
 <Requirement concern="Meters" id="3">
  <Constraint action="ensure" operator="with">
   <Requirement concern="G2S" id="1,2"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="Meters" id="3">
  <Constraint action="ensure" operator="with">
   <Requirement concern="SCP" id="1,2"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="G2S" id="7">
   <Constraint action="enforce" operator="xor">
   <Requirement concern="Proprietary Communication Protocol" id
       ="5"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="Error Condition" id="2,3">
  <Constraint action="provide" operator="for">
   <Requirement concern="G2S" id="4"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="G2S" id="3">
   <Constraint action="enforce" operator="xor">
   <Requirement concern="Proprietary Communication Protocol" id
       ="4"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="GameRecall" id="1,2">
  <Constraint action="ensure" operator="with">
   <Requirement concern="G2S" id="6"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="SCP" id="all">
   <Constraint action="enforce" operator="xor">
   <Requirement concern="Demo" id="all"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
<Composition>
 <Requirement concern="G2S" id="all">
   <Constraint action="enforce" operator="xor">
   <Requirement concern="Demo" id="all"/>
  </Constraint>
  <Outcome action="fulfilled"/>
 </Requirement>
</Composition>
```

Listing B.3: Interactions for the selected requirements expressed using MDSOCRE-i notation

```
        <Composition>
            <Requirement concern="G2S" id="1,2">
                <Interaction type="dependency">
                    <Requirement concern="Meters" id="3"/>
                </Interaction>
            </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="Proprietary Communication Protocol
                " id="1,2">
                <Interaction type="dependency">
                    <Requirement concern="Meters" id="3"/>
                </Interaction>
            </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="Proprietary Communication Protocol
                " id="5">
                <Interaction type="mutex">
                    <Requirement concern="G2S" id="7" />
                </Interaction>
            </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="Error Condition" id="2,3">
                <Interaction type="reinforcement" >
                    <Requirement concern="G2S" id="4"/>
                </Interaction>
            </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="Proprietary Communication Protocol
                " id="4">
                <Interaction type="mutex" >
                    <Requirement concern="G2S" id="3"/>
                </Interaction>
            </Requirement>
        </Composition>
  <Composition>
    <Requirement concern="G2S" id="6">
      <Interaction type="dependency">
       <Requirement concern="GameRecall" id="1,2"/>
      </Interaction>
    </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="Proprietary Communication Protocol
                " id="all">
                <Interaction type="conflict" >
                    <Requirement concern="Demo" id="all"/>
                </Interaction>
            </Requirement>
        </Composition>
        <Composition>
            <Requirement concern="G2S" id="all">
                <Interaction type="conflict" >
                    <Requirement concern="Demo" id="all"/>
                </Interaction>
            </Requirement>
        </Composition>
```

Appendix C

Resumen en Español

C.1 Motivación

La programación orientada a aspectos (AOP) provee herramientas poderosas para el encapsulamiento de las llamadas "incumbencias transversales" (*cross-cutting concerns*) a nivel de programación.

Las ideas y conceptos de AOP rápidamente impregnaron otras etapas del desarrollo de software, incluyendo el análisis de requerimientos y el diseño, dando origen respectivamente a la ingeniería de requerimientos orientada a aspectos (AORE, por sus iniciales en inglés) y modelado orientado a aspectos (AOM). El conjunto de técnicas orientadas a aspectos, aplicadas al ciclo desarrollo de software se conocen como *aspect oriented software development* (AOSD).

Dado que los aspectos encapsulan incumbencias independientes, se esperaría que no debieran interferir, pero es reconocido que dichas interferencias ocurren [58, 74, 88]. Estas interacciones son un tema abierto en la comunidad científica que realiza investigación en orientación a aspectos [18]. El mismo ha sido parcialmente estudiado acotándolo a etapas aisladas del desarrollo de software, pero las interacciones entre aspectos nunca han sido tratadas de manera consistente a largo del ciclo que va desde los requerimientos hasta la implementación. Por otra parte, los ejemplos de interacciones encontrados en la bibliografía refieren en general a un solo tipo de interacciones, que llamamos interacciones a nivel de *joinpoint*.

Esta tesis propone el estudio de las interacciones entre aspectos en tres etapas del desarrollo de software: análisis de requerimientos, diseño e implementación, en el contexto de un sistema de software complejo, de tipo industrial. Nuestra hipótesis es que el estudio de las interacciones desde etapas tempranas del desarrollo proveerá información importante sobre la naturaleza de las mismas, la cual permitirá desarrollar herramientas efectivas para su tratamiento.

C.2 Objetivos

Con el propósito de desarrollar mecanismos que permitan modelar e implementar las interacciones entre aspectos, en este trabajo se ha llevado adelante el desarrollo orientado a aspectos de un caso de estudio derivado de la industria, haciendo foco en el problema de las interacciones desde el inicio.

Podemos resumir los objetivos de este trabajo de la siguiente manera:

- El objetivo principal consiste en comprender las interacciones entre aspectos a partir de su estudio en varias etapas del ciclo de desarrollo, para proveer mecanismos o soluciones que permitan tratarlas de manera adecuada. A partir de este objetivo se desprenden los siguientes sub-objetivos.
- El análisis un caso de estudio industrial complejo con presencia de interacciones entre las diferentes incumbencias o aspectos.
- El estudio de la naturaleza de dichas interacciones y su clasificación de acuerdo a las taxonomías existentes.
- La evaluación de las capacidades expresivas de los enfoques de AORE existentes para el modelado de las interacciones, proponiendo extensiones cuando fuera necesario.
- Evaluación del soporte para interacciones provisto por enfoques de modelado orientado a aspectos, proponiendo extensiones cuando fuera necesario.
- La implementación de las interacciones, utilizando lenguajes orientados a aspectos dinámicos y estáticos, con el objetivo de evaluar el soporte existen proponiendo las extensiones que fueran necesarias.

C.3 Detalle del Contenido de la Tesis

Para el desarrollo de este trabajo hemos utilizado como caso de estudio el software para *slots machines* (SM, máquinas tragamonedas).

Las SM son máquinas de apuestas, usualmente instaladas en casinos. Por tratarse de máquinas que manejan grandes volúmenes de dinero tanto el hardware como el software están sujetos a numerosas reglamentaciones, estándares aplicables y recomendaciones técnicas. El software de las SM, cuyas características se presentan en el capítulo 3 se compone de numerosas incumbencias transversales funcionales (*functional crosscutting concerns*).

Estas incumbencias presentan varios ejemplos de interacciones, los cuales pueden ser clasificados de acuerdo a la taxonomía de Sanen *et al.* [74]. En dicha taxonomía las interacciones entre aspectos se clasifican en cuatro categorías:

Conflict: representa un interferencia semántica entre aspectos. Si existe un conflicto entre dos aspectos A y B, cada uno funciona correctamente sobre el sistema de base, pero los dos no pueden ser instalados al mismo tiempo. La combinación de los aspectos A y B genera comportamiento indeseable en el sistema.

Mutex: En este caso dos aspectos brindan una funcionalidad similar pero no pueden ser instalados en el sistema al mismo tiempo.

Dependency: El aspecto A *depende* del aspecto B, si A necesita que B este instalado para funcionar correctamente. Si B no esta presente, el comportamiento de A no será el esperado.

Reinforcement: Existe un *refuerzo* del aspecto B en el aspecto A si la presencia de B beneficia al aspecto A. El aspecto A puede funcionar en ausencia de B, pero la presencia de B habilita funcionalidad extra en el aspecto A.

Dada la importancia que tienen las interacciones en el dominio elegido (ver sección 3.8), hemos decido aplicar AOSD haciendo especial énfasis en la identificación, documentación, modelado e implementación de las interacciones. En las distintas etapas del ciclo de desarrollo se ha analizado el soporte existente, extendiendo algunos de los enfoques para AORE, AOM y AOP con soporte explícito para las interacciones.

C.3.1 Interacciones entre Aspectos en el Análisis de Requerimientos

Para este trabajo se ha comenzado en etapas tempranas del desarrollo, particularmente en el análisis de requerimientos. De acuerdo a Liu *et al.* [57] la mayoría de las interacciones entre características (*feature interactions*) pueden ser detectadas en etapas tempranas del ciclo de desarrollo razonando acerca de las causas y construyendo modelos de ellas. El objetivo de esta detección es documentar tantas interacciones como sea posible de manera que esta información pueda ser utilizada en etapas posteriores (diseño e implementación).

El resultado de esta etapa debe ser un modelo de requerimientos tan consistente como sea posible. Dado que una consistencia total y absoluta no es posible ante la presencia de conflictos, estos deben ser documentados de forma de diferir su resolución a las etapas posteriores del desarrollo. Para lograr este objetivo es necesario apoyarse en mecanismos expresivos provistos por las técnicas en seleccionadas para esta etapa.

A nuestro mejor entender, no existe ningún trabajo previo detallando experiencias acerca del soporte de interacciones en metodologías AORE, en el contexto de un caso industrial. Por lo tanto, se decidió realizar un estudio en profundidad utilizando dos reconocidos enfoques de AORE, con el fin de evaluar su aplicabilidad en el dominio de las SM: Theme/Doc [9] y MDSOCRE [60].

En el capítulo 4 se muestra cómo estos enfoques fueron aplicados. Los resultados han demostrado deficiencias, como ser la falta de granularidad en Theme/Doc y la ausencia de soporte de interacciones para ambos. En el mismo capítulo se han propuesto extensiones para la notación de Theme/Doc, agregando granularidad (permite indicar que requerimientos intervienen en las relaciones entre concerns) y la capacidad de expresar requerimientos derivados (ver sección 4.4.1). En el caso de MDSOCRE se agregó soporte explícito para interacciones, el cual fue testeado mediante un experimento con ingenieros expertos en el dominio y demostró ser más preciso para la representación de interacciones y más eficiente en términos de tiempo (ver sección 4.5).

C.3.2 Interacciones entre Aspectos en el Diseño

El siguiente paso consiste en modelar el software utilizando un enfoque adecuado de AOM (*aspect oriented modeling*). En la fase de diseño el objetivo es refinar la especificación de requerimientos para obtener un modelo de los artefactos de software que conformarán el sistema final. Este modelo, descripto en un documento, debería ser utilizado por los desarrolladores como guía durante la etapa de implementación de manera relativamente independiente. Por lo tanto debe ser lo suficientemente completo como para que los desarrolladores no necesiten releer anexos conteniendo requerimientos. En esta fase, esperamos producir diseños completos sin necesidad de recurrir a (muchos) documentos adicionales con notaciones *ad-hoc*, ya que en este caso utilizar una metodología AOM reportaría muy pocos beneficios, y consideraríamos desarrollar nuestro propio enfoque AOM. En particular nuestras expectativas incluyen el soporte explícito para interacciones entre concerns, y la mantenibilidad.

Es conocido que la presencia de aspectos en un sistema que evoluciona puede ser problemática [51]. Estos problemas deben mitigarse gracias a la información explícita disponible en el diseño. Las interacciones entre aspectos pueden complicar la comprensión sobre el comportamiento esperado del sistema. Esta información es crucial para la implementación correcta del sistema y su evolución. Por lo tanto la documentación de las decisiones de diseño debe incluir no sólo los módulos que serán aspectos y donde atraviesan a otros módulos, sino también cómo interactuan entre ellos. Esta información debe disponerse de forma explícita, de manera que pueda ser comunicada adecuadamente a la etapa de implementación. El enfoque AOM utilizado debe proveer soporte para expresar dicha información.

Hasta donde conocemos, no existe ningún trabajo que evalúe los enfoques de AOM en un contexto industrial con foco en las interacciones entre concerns. Nuevamente decidimos realizar una evaluación de dos reconocidos enfoques para *aspect oriented modeling*: Theme/UML [26] y WEAVR [28]. En el capítulo 5 se estudió cómo expresar a nivel de diseño el comportamiento que las SM deben proveer, incluyendo las interacciones entre aspectos. Si bien Theme/UML afirma brindar soporte para conflictos, hemos comprobado que el soporte esta orientado a problemas menores como conflictos de cardinalidades o visibilidad debido a declaraciones diferentes en distintos *temas*. Por otra parte, el soporte de WEAVR es más avanzado, permitiendo resolver a nivel de *joinpoint* las dependencias y conflictos. Sin embargo, también es insuficiente ya que no permite expresar de manera explícita las interacciones que ocurren en el dominio de las SM. En el mismo capítulo se ha mostrado que es posible para ambos enfoques describir el comportamiento de las interacciones de forma implícita, lo cual acarrea problemas de mantenimiento y evolución en el sistema.

En ambos casos existen problemas de escalabilidad. En el caso de Theme/UML no existe una herramienta que soporte la metodología. En el caso de WEAVR si bien existe una herramienta reportada en al bibliografía, la notación no soporta abstracciones que permitan aplicar soluciones a problemas recurrentes (como las interacciones).

C.3.3 Interacciones entre Aspectos en la Implementación

Una vez que las decisiones de diseño respecto de las interacciones han sido documentadas, debe realizarse la implementación de las mismas. En esta etapa es deseable que el programador disponga de construcciones del lenguaje que le permitan codificar de manera explícita las interacciones. En su defecto, será necesario implementar las interacciones valiendose de construcciones definidas enel lenguaje, las cuales no fueron pensadas para esta función. Por lo tanto, se decidió implementar las interacciones utilizando lenguajes orientados a aspectos de propósito general, con el objetivo de evaluar el impacto que ellos tienen en las interacciones entre aspectos. En primer lugar se eligió a AspectJ [52] que es, discutiblemente, el lenguaje orientado a aspectos más maduro e influyente. Dado que AspectJ es un lenguaje estático, hemos decido contrastar los resultados con la implementación de las interacciones en un lenguaje dinámico, en este caso hemos elegido a PHANtom [34].

La programación utilizando tales lenguajes es una tarea importante para poder evaluar la conveniencia de cada uno a la hora de implementar las interacciones mediante lógica específica para cada caso, y definir cuales son las extensiones más necesarias para soportarlas de manera nativa. En el capítulo 6 se describe la forma en la cual cada una de las interacciones pueden implementarse en cada uno de los lenguajes, analizando el impacto que estos tienen en dicha implementación. A partir de este trabajo se puede ver que de los cuatro tipos de interacciones sólo *reinforcement* se benefició de los chequeos en tiempo de compilación que realiza AspectJ, debido al uso de *inter-type declarations. Dependency* y *mutex* ocurren de manera similar, y requieren el uso de lógica *ad-hoc* para su implementación. En el caso de *dependency* es deseable añadir chequeos en run-time para asegurar la consistencia del sistema (ver sección 7.3.4). Por otro lado, *conflict* se beneficia del *weaving* dinámico en PHANtom y requiere la instrumentación de los aspectos (para desactivarlos) en el caso de AspectJ. Basados la experiencia de esta implementación, se propusieron extensiones a AspectJ en la sección 6.6, las cuales permiten tratar tanto interacciones a nivel de *joinpoint* como interacciones semánticas. Estas extensiones pueden ser implementadas como una combinación de chequeos en tiempo de compilación y ejecución.

C.4 Contribuciones

Las contribuciones de esta tesis se resumen a continuación:

- El reporte de un ciclo de desarrollo orientado a aspectos para un dominio industrial. Hasta donde conocemos no existe un reporte incluyendo análisis, diseño e implementación de un sistema industrial utilizando orientación a aspectos.

- Dos reconocidos enfoques de AORE fueron evaluados y extendidos. Estas extensiones fueron evaluadas experimentalmente con el fin de juzgar su aplicabilidad. Este trabajo fue publicado en [97, 98, 99].

- Dos enfoques de AOM, que afirman tener soporte para interacciones, han sido evaluados utilizando las interacciones encontradas en el dominio de las SM. En ambos casos se han encontrado falencias, las cuales han sido reportadas y publicadas en [36][1].

- La implementación de los 4 tipos de interacciones utilizando tanto un lenguaje de aspectos dinámico como uno estático. Los resultados muestran la naturaleza dinámica o estática del lenguaje usado tiene un impacto considerable en la implementación de la misma. A partir de la experiencia se propone una extensión para AspectJ que cubre las interacciones

[1]Este artículo fue premiado como *best paper* en el track industrial de MODELS 2011.

encontradas y además las reportadas en la bibliografía. Estos resultados no se encuentran publicados aún pero están inspirados en nuestro trabajo reportado en [100].

- Finalmente, esta tesis contribuye el estudio en profundidad de un caso industrial y complejo, compuesto por numerosas incumbencias transversales, muchas de ellas funcionales. Ejemplos genuinos y nuevos de interacciones han sido encontrados y documentados. Estas interacciones difieren de los reportados en la bibliografía disponible y por lo tanto presentan un desafío relevante para estos y otros enfoques de orientación a aspectos.

C.5 Análisis de las conclusiones

Durante nuestra investigación hemos observado que en general las interacciones entre aspectos, especialmente las semánticas, han sido parcialmente ignoradas. Aun cuando algunas interacciones a nivel de *joinpoint* son soportadas por algunos de los trabajos relacionados no existe ningún reporte de un desarrollo de un sistema industrial y complejo, que valide su aplicabilidad.

Hemos comprobado la existencia de numerosas interacciones semánticas entre *concerns* (o aspectos, dependiendo de la etapa del desarrollo de la que estemos hablando) en el dominio de las SM. Estas interacciones demostraron ser diferentes a las habitualmente tratadas en la bibliografía. Por lo tanto, no es sorprendente que los enfoques de desarrollo orientados a aspectos muestren limitaciones y deficiencias para expresarlas correctamente. Por lo tanto, en cada etapa del ciclo de desarrollo se ha observado que las metodologías y enfoques orientados a aspectos carecen de la madurez suficiente para el desarrollo de sistemas complejos donde existan interacciones entre aspectos.

Consideramos que este soporte debe ser agregado a los enfoques orientados a aspectos, ya que ignorar la existencia de las interacciones restringe notablemente la aplicabilidad de la orientación a aspectos. En este sentido hemos presentado un soporte explícito para interacciones en los enfoques AORE estudiados. Los experimentos realizados demostraron que las extensiones propuestas permiten expresar de forma eficaz la semántica de la interacciones (sección 4.5). Además estas extensiones permiten a los ingenieros interpretar las relaciones ente los concerns en forma más rápida. En la etapa de diseño se ha mostrado cómo el comportamiento referido a las interacciones puede expresarse sólo de manera implícita debido a la ausencia de soporte adecuado para las interacciones. Las limitaciones en cuanto a expresividad y escalabilidad de los enfoques evaluados permitarán desarrollar extensiones para las interacciones. En la etapa de implementación, dos lenguajes orientados a aspectos de propósito general fueron utilizados mostrando que es necesario soporte explícito para las interacciones. A partir de estas implementaciones es posible concluir qué efecto tiene la dinamicidad del lenguaje elegido, en la implementacion de los cuatro tipos de interacciones tratadas. Estos efectos pueden ser tenidos en cuenta al momento de elegir un lenguaje para la implementación, considerando las interacciones presentes. Finalmente, se presentaron extensiones para el lenguaje AspectJ, las cuales permiten tratar las interacciones a nivel de *joinpoint* (como las reportadas en la bibliografía) y semántico.

Como una reflexión final respecto de las interacciones, podemos concluir que las interacciones tratadas en etapas tempranas del desarrollo tienen una forma más genérica. Por lo tanto, es más sencillo proveer soporte para ellas. En etapas posteriores del desarrollo, las decisiones de diseño e implementación referidas a las interacciones, requieren mecanismo con una semántica muy particular.

C.5.1 Evaluación de la Orientación a Aspectos en un Dominio Industrial

El caso de estudio analizado en esta tesis nos ha permitido, además de evaluar el soporte de interacciones y proponer extensiones para soportarlas en las distintas etapas, juzgar la aplicabilidad de los enfoques orientados a aspectos a problemas reales en la industria. Hemos visto que en la etapa de requerimientos no proveen soporte para interacciones, a excepción de los conflictos. Sin embargo, dado que el signicado de "conflicto" difiere entre los enfoques, también lo hace el soporte que ellos proveen. En general los enfoques de AORE que soportan conflictos permiten detectarlos o proveen herramientas para eliminarlos. Vemos que este soporte es insuficiente, ya que algunos conflictos deben ser documentados para su tratamiento en posteriores etapas del ciclo de desarrollo.

Las herramientas AOM se encuentran en un estado de inmadurez mayor que las de la etapa análisis de requerimientos e implementación. Hemos observado que cuentan con tres problemas importantes: 1) soporte pobre para interacciones, 2) notaciones incongruentes para conceptos comunes de la orientación a aspectos y 3) falta de escalabilidad debido a la ausencia de herramientas. Con respecto a 1) hemos observado que debido a que proveen un soporte pobre para interacciones, información relevante debe ser expresada implícitamente. En lo que respecta a 2) las diferentes notaciones, radicalmente diferentes, son un factor distractivo que agrega complejidad innecesaria. En nuestra opinión la notación, al menos para los conceptos básicos de AOP como ser: *pointcut*, *advice*, aspecto, introducciones (*inter type declarations*) etc, debería ser unificada. Finalmente, con respecto a 3) vemos que los enfoques de AOM y AORE sufren un problema de falta de escalabilidad, directamente relacionado con la escacez de herramientas que automaticen o den soporte a ciertos procesos. Las herramientas para AOM pueden proveer soporte para el modelado, el weaving y la generación de código. La falta herramientas para modelado puede ser parcialmente subsanada con la utilización de abstracciones que permitan aplicar la misma solución a diferentes escenarios, de forma tal de reducir la cantidad de documentos de diseño a generar y mantener. Sin embargo, no hay posibilidad de aliviar la falta de herramientas para el *weaving* automático de modelos. Consideramos que tanto las herramientas para modelado y *weaving*, a diferencia de las herramientas para generación de código, son críticas para el éxito de AOM.

En lo que respecta a la etapa de implementación, vemos que AOP presenta un reto para los programadores a la hora de comprender el comportamiento del sistema, ya que la naturaleza de los aspectos hace que modifiquen el comportamiento de muchos módulos. Por lo tanto, la ayuda provista por el lenguaje y las herramientas de desarrollo para la correcta visualización de los efectos de los aspectos es crucial. Las interacciones entre aspectos complican aún más el panorama, dificultando comprender el comportamiento de sistema final. Hemos observado que los lenguajes no proveen un soporte adecuado para explícitas las interacciones, haciendo necesario implementarlas de manera *ad-hoc*, Esto resulta en restricciones del comportamiento implícitas, lo cual acarrea problemas de mantenibilidad.

Creemos que varios de los puntos mencionados pueden ser las causas por las cuales la orientación a aspectos no se ha convertido en un paradigma de aplicación masiva. A partir de estas observaciones podemos afirmar que AOSD no está listo para la industria.

C.6 Trabajo Futuro

A partir del trabajo desarrollado para esta tesis se desprenden las siguientes líneas de acción a futuro:

- El trabajo desarrollado nos ha mostrado que existen muchas interacciones en el dominio de las SM y que si bien se pueden clasificar de acuerdo a taxonomías conocidas, la forma de las interacciones es diferente a las reportadas en la bibliografía. Esto plantea un nuevo interrogante: ¿estas interacciones únicamente ocurren en el dominio de las SM? ¿O no han sido reportadas como consecuencia de una selección tendenciosa de los casos de estudio? Es necesario estudiar otros dominios complejos para responder a esta pregunta.

- La trazabilidad es un elemento importante en el proceso de desarrollo de software. Las interacciones deberían ser trazadas desde los requerimientos hasta la implementación, de forma tal que al evolucionar los requerimientos asociados a una incumbencia que presenta interacciones, las restricciones asociadas a la interacción no sean violadas. De aquí se desprende que es necesario estudiar y posiblemente realizar nuevas extensiones que permitan dicha trazabilidad.

- Motivados por las diferentes fuentes de requerimientos, en la etapa AORE proponemos el estudio del impacto de nuestras extensiones en la evolución de los requerimientos (ver sección 7.3.2).

- En la etapa de diseño es necesario proveer soporte explícito para interacciones a las metodologías utilizadas. En la sección 7.3.3 se discuten brevemente estas posibles extensiones. Además nuevos enfoques de AOM reportan tener un soporte avanzado para interacciones, en particular, la aplicación de RAM [55] es un de los trabajos planeados a corto plazo.

- En la etapa de implementación, en la sección 7.3.4 se describe brevemente como como las configuraciones de *deployment*[2] del sistema afectan o ayudan a implementar las interacciones, y como chequeos en *run-time* deberían ser realizados para asegurar la consistencia del sistema. Diferentes experimentos deben ser realizados para luego comparar la implementación de las interacciones usando estos mecanismos con las implementaciones ya realizadas.

- También en la etapa de implementación es necesario evaluar las extensiones propuestas para AspectJ. En este caso deben considerarse no sólo

[2]Llamamos configuración de *deployment* al conjunto de aspectos considerados activos para un *deployment* del sistema.

las interacciones encontradas en el dominio de las SM, sino también aquellas reportadas en la literatura sobre interacciones entre aspectos, las cuales en su mayoría son interacciones a nivel de *joinpoint*.