Expressing Aspectual Interactions in Design: Experiences in the Slot Machine Domain

Johan Fabry[1,⋆], Arturo Zambrano[2], and Silvia Gordillo[2]

[1] PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://pleiad.cl
[2] LIFIA, Facultad de Informática, Universidad Nacional de La Plata
La Plata, Argentina

Abstract. In the context of an industrial project we are implementing the software of a casino slot machine. This software has a significant amount of cross-cutting concerns that depend on, and interact with each other, as well as with the modular concerns. We therefore wish to express our design using an appropriate Aspect-Oriented Modeling methodology and notation. We evaluated two of the most mature methodologies: Theme/UML and WEAVR, to establish their suitability. Remarkably, neither of these allow us to express any of the dependencies and interactions to our satisfaction. In both cases, half of the interaction types cannot be expressed at all while the other half need to be expressed using a workaround that hides the intention of the design. As a result, we consider both methodologies and notations unsuitable for expressing the dependencies and interactions present in the slot machine domain. In this paper we describe our evaluation experience.

1 Introduction

A slot machine (SM) is a casino gambling device that has five *reels* which spin when a *play* button is pressed. An SM includes some means for entering money, which is mapped to *credits*. The player bets an amount of credits on each play, the SM randomly selects the displayed symbol for each reel, and pays the corresponding prize, if any. Credits can be extracted (called a *cashout*) by different mechanisms such as coins, tickets or electronic transfers.

In the context of an industrial project we were required to re-implement the software for a particular SM. Previous experience had taught us that, beyond the main functionality sketched above, there are a significant amount of cross-cutting concerns present in such applications. For example: counters need to be maintained throughout the application to be able to audit the SM, and the complete working of the SM needs to be accessible over the network. Moreover, these concerns depend on, and interact with each other as well as with

⋆ Partially funded by FONDECYT project 1090083.

J. Whittle, T. Clark, and T. Kühne (Eds.): MODELS 2011, LNCS 6981, pp. 93–107, 2011.

the modularized concerns. We therefore opted to use Aspect-Oriented Software Development in this implementation, taking special care of dependencies and interactions between the different aspects and modules. In a previous step, we analyzed the different concerns that define the behavior of SMs, with a specific focus on concern interactions at the requirements level [13].

The second step in our development process 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 article.

As basis for our selection we used surveys on AOM [2,12], complemented by a study of more recent literature. The chosen approaches are Theme/UML [4] and WEAVR [6,5]. 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 requirements methodology for requirements specification [13]. WEAVR is arguably the best-known industrial application of AOM, and the only methodology that we are aware of that is used in industry to develop complex applications.

We now give an overview of the requirements we have for the design document, before giving a high-level overview of the design and the different interactions that need to be specified. Section 4 then proceeds with an evaluation of Theme/UML, and Sect. 5 follows up with an evaluation of WEAVR. We present related work in Sect. 6, and conclusions and future work in Sect. 7.

2 Requirements for the Design

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 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 being evolved can be problematic [9]. 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 performed previously [13]. 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.

3 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.

3.1 Aspects in the Design

A class diagram that shows the outline of the design is given in Fig. 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, S Communications Protocol, G2S Protocol. We give an overview of these aspects next.

Metering. The Metering aspect crosscuts Game and other base entities in order to keep meters data up to date. Meters are essentially a set of counters that keep information about past plays, *e.g.* the total amount bet. This information is used, among other things, to create reports.

Demo. For legal certification the SM must have a 'Demo' mode, where all possible outcomes for a play can be simulated. The Demo concern therefore needs to control the outcome produced by the Game class. It furthermore crosscuts Metering to avoid polluting accounting meters when it is active.

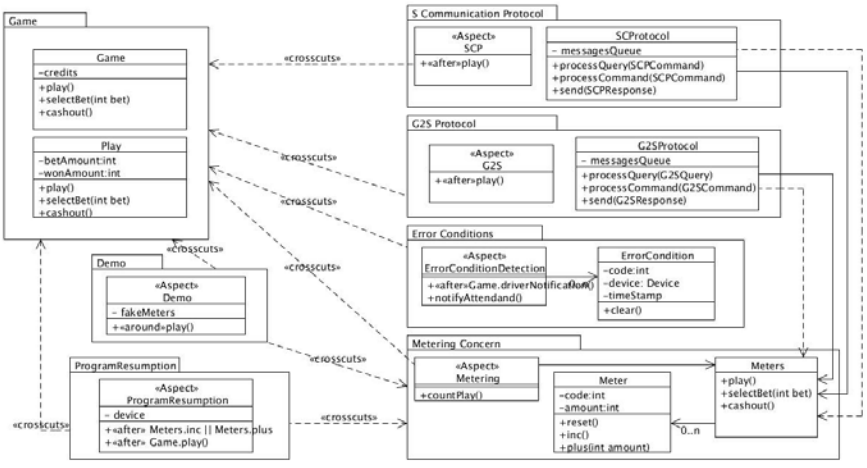


Fig. 1. Overview of the class structure of the design

Program Resumption. Program Resumption is a persistence and recovery requirement. The system should recover the last state after a power outage. Information to be saved includes the status of the current play and the values of the meters.

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.

Communication Protocols. The S Communications Protocol[1] (SCP) and G2S Protocol are communications protocols frequently used in the gaming industry. Their corresponding 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 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 Fig. 1.

[1] A pseudonym, licensing restrictions prohibit us from using the real name.

3.2 Interactions between Concerns

Our resulting design document not only needs to contain the information of the aspects present in the system, but also how they crosscut. It is also necessary that the interactions which were identified in the requirements analysis phase be present in the design document. To better understand what our needs are for this part of the design document, we now give an overview of the different interactions in the SM, and how we want this to be reflected in the document.

We structure this discussion and the evaluations of the AOM approaches later in the text using the AOSD-Europe technical report on interactions [11]. It classifies interactions in four different types: *dependency*, *conflict*, *mutex* and *reinforcement*, and the SM software contains an instance of each of these types.

Conflict: Demo versus Multiple Concerns. The aspects of Meters, Communication Protocols and Program Resumption 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 aspects. This is as the legislation states that a play in Demo mode must not alter the meters nor that its activity is visible over the network. Hence, after a Demo session the Game must recover its original status and any event or state change while in Demo must not be reported by the communication protocols.

In order to cope with this conflicting behavior the design and implementation must provide support for:

- Avoiding simultaneous activation of Demo and other conflicting aspects.
- While being in Demo a fake set of meters should be used. This ensures that actions in demo mode do not alter the meter values of normal operations.

Communication Protocols: Mutex, Reinforcement and Dependency. Both communication protocols provide similar functionality: allowing the server to query information and set some configuration values and state on the SM. For read-only behavior, such as reporting the value of a meter, there is no problem with having them active at the same time as no interference will result. On the other hand, for operations that alter the state of the machine, mutual exclusion must be ensured during a single program execution. If not, inconsistencies in the SM 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 timing of events on the SM is ambiguous. To document this *mutex* what we need is the ability to express that certain object interactions may not occur during the programs' execution.

There is a *reinforcement* from Error Conditions to Communication Protocols. Not all the Error Conditions specified in the legislation 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, the communication protocols must be able to report this to the server. This means that during development of new versions of the Game, when new error conditions are present, the associated behavior in the Communication Protocols should be revisited to ensure that the

new information is properly reported. Hence we need to document that a change in a different part of the application enables optional or extended behavior of a given concern.

The last interaction regarding Communication Protocols is a *dependency* of them on Meters. The protocols access the meters in order to report their values to the server. Consequently, if for some reason metering is not present, the communication protocols cannot operate. When a communication protocol is enabled, meters must be present, and must be properly fed. We need to document this dependency to ensure the consistent behavior of the system.

4 Evaluation of Theme/UML

Theme/UML is the second half of the Theme approach for Aspect-Oriented requirements analysis and design. The first half is called Theme/Doc and is a methodology for AO requirements analysis. Theme provides for 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 [13] we have evaluated Theme/Doc, and now continue with an evaluation of Theme/UML.

The Theme/UML approach [4] 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.

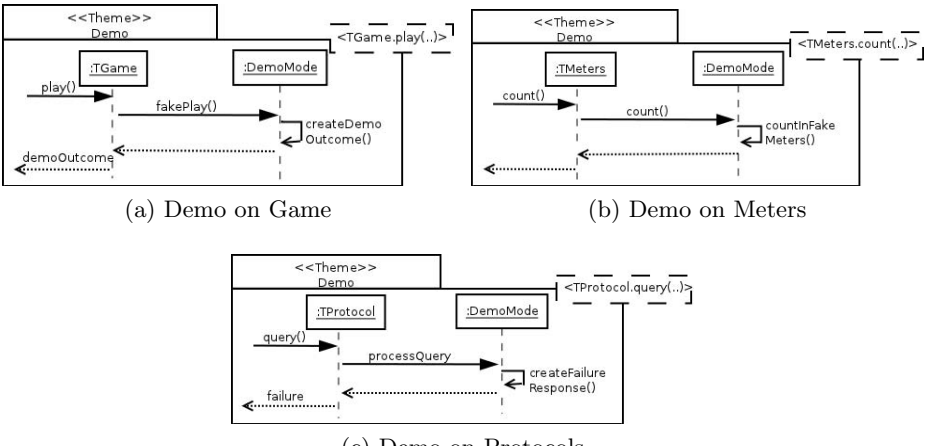
In accordance to Fig. 1, we modeled Game as a base theme and Demo, G2S, Meters, and SCP as crosscutting themes. We found it is straightforward to express where to attach the crosscutting behavior, both on the base themes and on other crosscutting themes. However, when considering interactions we find that Theme/UML does not perform as well. We now discuss the obstacles we encountered classified in the four different kinds of interactions we discussed in Sect. 3.2: Conflict, Mutex, Reinforcement and Dependency.

4.1 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 3.2, 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, making the conflict implicit.

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 are captured and skipped. Considering the communication protocols, their original behavior is altered: instead of responding to queries, failure responses are returned.



(a) Demo on Game

(b) Demo on Meters

(c) Demo on Protocols

Fig. 2. The Demo theme affecting the behavior defined in Game, Metering and Protocols

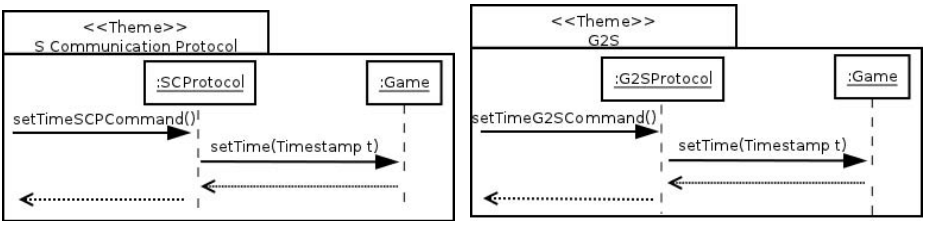
Our model is shown in Fig. 2. We use Theme/UML sequence diagrams, a straightforward extension of UML sequence diagrams. The figure shows three Themes, each of which has a template parameter in the top right corner, corresponding to the message send that starts the sequence. At composition time, this 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. 2 there are no _do_play, _do_count or _do_query calls, which means the join point behavior is skipped.

The above solution has the major downside that 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. As we require that the design intent is explicit, we do not consider this a feasible solution.

4.2 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 that multiple protocols set 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.



(a) SCP theme (b) G2S theme

Fig. 3. Two themes configuring the same item in the Game

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. 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 way in which we can express this mutex relationship between both sequence diagrams. Neither do we see an 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.

4.3 Reinforcement

The error condition aspect reinforces the behavior of the communication protocols, reporting all 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 but 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.

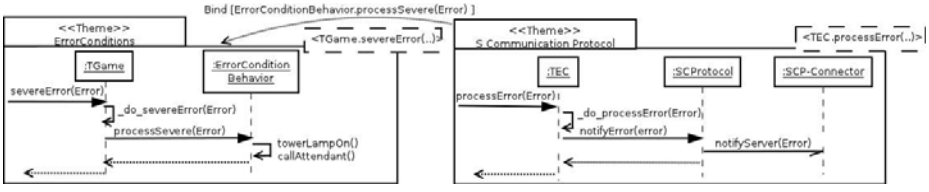


Fig. 4. SCP Theme reinforced by Error Conditions theme

The left hand side of Fig. 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. 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 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.

4.4 Dependency

The metering theme maintains track of given events in the game by changing the values of meters objects. Complementary to this, the communication protocols themes specify that to respond to queries sent by the remote server, the information stored in the meters objects are 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 [4]. 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.

What we need is a way to express that the meters themes are 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.

4.5 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. As a result, we consider Theme/UML inappropriate to specify the design of a SM.

5 Evaluation of WEAVR

WEAVR is an add-in extension to the MDE tool suite used by Motorola, adding support for AOM to their process of building telecom software [6,5]. 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.

Next to a UML notation, the Motorola tool suite also uses SDL [7] 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++.

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 [6,5]. 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.

5.1 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

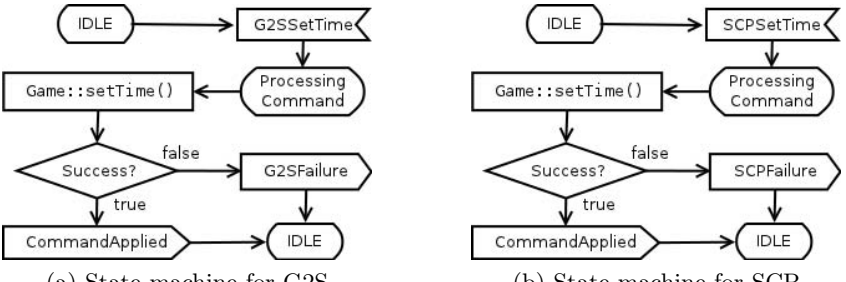
join point. The relationship states that the aspect that is hidden does not apply in those cases. For example, specifying AspectA hidden_by AspectB denotes that at a join point captured by both aspects, only the behavior of AspectB will be executed. In other words, we can state that the presence of one 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 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 SProtocol.

Similar to the workaround for Theme we proposed in Sect. 4.1, we can provide a design that incorporates the required conflict resolution behavior. Advice in WEAVR are always around advice, and use a proceed call. As in Fig. 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. 4.1, most importantly the loss of the explicit conflict specification.

5.2 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 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. 5a executes if the state machine of Fig. 5b was previously run, and vice-versa. However, WEAVR does not provide for any way in which this can be specified.



(a) State machine for G2S (b) State machine for SCP

Fig. 5. Mutually exclusive state machines for the setTime command

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 are 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.

5.3 Reinforcement

The design of the reinforcement from error conditions to communication protocols is similar to the design in Theme/UML discussed in Sect. 4.3. 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 Sect. 4.3, 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 Dependency

Similar to the design in Theme, shown in Sect. 4.4, we have an interplay between the metering concern and the communication concerns. The metering concern capturing events regarding game activity and updating the meters, while the communication protocols consult data contained in these meters when processing server requests. In Fig. 6, 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 due to lack of space). 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

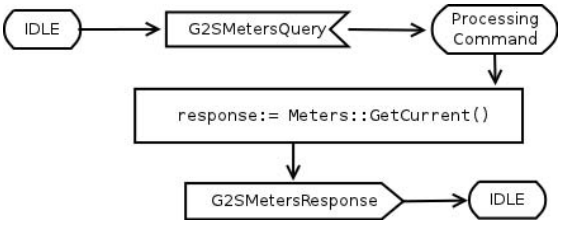


Fig. 6. Part of the G2S Protocol state machine depending on meters

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. 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.

5.5 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 a SM.

6 Related Work

Schauerhuber *et al.* authored a survey of AOM approaches [12] 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.* [3] and Mehner *et al.* [10].

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 [1] 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 have been detected during the requirements phase [13]. Instead of detection, we need for 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 [8]. In our case however some interactions are required to obtain the desired behavior, and other interactions cannot be removed but should be controlled instead.

It is interesting 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 our context, and we see no reason why it would be an exceptional case.

7 Conclusions and Future Work

The AOSD-Europe technical report on interactions [11] 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 our study is that neither Theme/UML nor WEAVR allow us to satisfactorily express any of the four types of dependency. This although both approaches are considered mature, are accepted by the community, and furthermore claim to have support for specific kinds of interactions. 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.

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.

References

1. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In: ISSRE, pp. 161–170. IEEE Computer Society, Los Alamitos (2010)

2. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of analysis and design approaches. Tech. Rep. AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, University of Lancaster (2005)
3. Ciraci, S., Havinga, W., Aksit, M., Bockisch, C., van den Broek, P.: A graph-based aspect interference detection approach for uml-based aspect-oriented models. T. Aspect-Oriented Software Development 7, 321–374 (2010)
4. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Object Technology Series. Addison-Wesley, Boston (2005), `http://fparreiras/books/AspectOrientedAnalysisAndDesign.chm`
5. Cottenier, T., van den Berg, A., Elrad, T.: Motorola weavr: Aspect and model-driven engineering. Journal of Object Technology 6(7), 51–88 (2007)
6. Cottenier, T., Berg, A.V., Elrad, T.: The Motorola WEAVR: Model Weaving in a Large Industrial Context. In: Proceedings of the International Conference on AspectOriented Software Development, Industry Track (2006)
7. ITU, Z.: Specification and description language (sdl). In: International Telecommunication Union (2000)
8. Katz, E., Katz, S.: Incremental analysis of interference among aspects. In: Clifton, C. (ed.) FOAL, pp. 29–38. ACM, New York (2008)
9. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the evolution of aspect-oriented software with model-based pointcuts. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 501–525. Springer, Heidelberg (2006)
10. Mehner, K., Monga, M., Taentzer, G.: Interaction analysis in aspect-oriented models. In: RE, pp. 66–75. IEEE Computer Society, Los Alamitos (2006)
11. Sanen, F., Truyen, E., Win, B.D., Joosen, W., Loughran, N., Coulson, G., Rashid, A., Nedos, A., Jackson, A., Clarke, S.: Study on interaction issues. Tech. Rep. AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven (2006)
12. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer, M.: A survey on aspect-oriented modeling approaches. Tech. rep., Vienna University of Technology (2007)
13. Zambrano, A., Fabry, J., Jacobson, G., Gordillo, S.: Expressing aspectual interactions in requirements engineering: experiences in the slot machine domain. In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010), pp. 2161–2168. ACM Press, New York (2010)