Mapping Use Case Level Aspects to ObjectTeams/Java

Stephan Herrmann* Christine Hundt* Katharina Mehner*
Technical University Berlin
{stephan,resix,mehner}@cs.tu-berlin.de

ABSTRACT

Aspect-Oriented Software Development aims at supporting separation of crosscutting concerns throughout the full software lifecycle. In this contribution we focus on lifecycle support for crosscutting concerns with internal structure and complex behaviour. In order to make transitions between phases more seamless, support for such concerns is needed in all phases. In the past the programming language ObjectTeams/Java *has been developed which supports encapsulation of role-based collaborations and therefore is a suitable target platform for complex crosscutting concerns. We demonstrate how to develop requirements, analysis, and design models for this target language.*

1. INTRODUCTION

Aspect-oriented software development (AOSD) aims to support the *separation* of *concerns* which would be *crosscutting* with respect to pure object-oriented modularisation structures. As an extension of object-orientation AOSD provides new structuring constructs to avoid *tangling* of unrelated concerns and to prevent *scattering* of one concern over several locations. These advanced modularisation techniques allow for encapsulation of concerns in so called *aspects*.

The majority of existing AOSD approaches mainly address the implementation of aspects by focusing on aspect-oriented programming languages. As concerns emerge already in the earlier phases of the development process, it is desirable to provide aspects in requirements modeling, analysis and design as well.

In well-known aspect-oriented programming languages, such as AspectJ [9], aspects tend to suffer from some restrictions when comparing to regular classes. These restrictions mainly affect the ability to instantiate (and activate) aspects programmatically and limitations to the use of inheritance and polymorphism. Thus, aspects are not full-blown first class programming entities. To provide flexible instantiation and activation of aspects and to provide encapsulation and specialisation of collaborating aspects we have developed the programming language *ObjectTeams/Java* [7, 11]. This language decorates Java classes with *roles* which serve to implement potentially crosscutting concerns. Role behaviour is intrinsically bound to class behaviour by intercepting method calls to Java objects and by forwarding role instance methods to objects. A collaboration of roles is encapsulated in a package-like structure called *team*, which also has all properties of a class.

Since the ObjectTeams/Java language has matured and since we are starting to use it in a non-academic context [17] we are now aiming at full lifecycle support. In this vein, our first step has been to provide design support by means of an extension to UML class diagrams called *UML for Aspects* (UFA) which allows the modeling of the ObjectTeams/Java relevant features [6]. Also, an extension to UML sequence diagrams has been developed which shows role and class interactions [3].

While recent publications provide considerable advances of AOSD towards earlier phases of the software lifecycle [12, 13], we still see two issues that need to be addressed for full lifecycle support for aspects.

1. Seamlessness requires that isolated proposals for aspect oriented modeling are integrated into a continuous method.
2. Relevant use cases may need refinement in later phases yielding structured sub-models. When some use cases are identified as aspects a model of aspects is needed that specifically supports structural and behavioural refinement.

Regarding the first issue, in this paper we sketch a transition from an aspect-oriented use case model towards an aspect-oriented design, which seems to be the major remaining gap in the life-cycle.

When projecting typical non-functional aspects to the analysis level, some of these aspects shrink to a single sentence like: "Objects of class Foo are persistent", which can easily be modeled by a tagged value in the UML model, requiring no further modeling support. Complexity of such concerns is introduced only during later phases. Identifying use cases as aspects mainly reveals those aspects that bear more internal complexity, like workflows or collaborations of several communicating entities. ObjectTeams/Java specifically supports such collaborations, which is why we believe that it has the potential for the desired full-lifecycle seamlessness.

This paper reviews some recent work regarding use cases and aspects in section 2.1. We outline our process in section 2.2.

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

Section 3 illustrates the requirements elicitation of an aspect and the mapping into a design for ObjectTeams/Java using a simple example. We discuss emerging lifecycle support for AOSD in section 4 and conclude in section 5.

2. A USE CASE DRIVEN APPROACH

2.1 Background

Different approaches have been proposed to model crosscutting concerns on the use case level.

Sousa et al. [16] propose to model crosscutting concerns as use cases and provide a new relationship between use cases which is called ≪crosscut≫. This relationship defines a condition for the extension, the affected points in the execution of the affected use case in terms of its steps, and a so called composition rule operator. The operator follows the ideas of AspectJ by distingushing after, before, override and wrapping mode of composition. We will refer to the composition rule operator simply as *mode*. For an example of this notation refer to figure 3.

Unlike the invasive nature of the ≪extend≫ relationship[1], this approach uses a non-invasive specification of joinpoints rsp. pointcuts. The places in the execution which trigger the execution of the crosscutting use cases are only defined in the crosscut but not in the basic use case.

The authors provide the following heuristic to distinguish between the ≪include≫ relationship, the ≪extend≫ relationship and the newly defined ≪crosscut≫ (assume for the heuristic that use cases A, B are given):

> If the execution of the use case B represents a course that needs to be applied in the use case A, but (i) the use case A does not depend on the execution of the use case B to accomplish its primary goal; and (ii) the use case B is not a specific course of the use case A and therefore can be applied in other use cases; the use case B *crosscuts* the use case A [16].

AspectU [15] also provides an aspect-oriented extension to use cases. It introduces an aspect entity which is structured similarly to an AspectJ aspect consisting of pointcuts and advices.

The pointcut language matches steps from the use cases and provides binding predicates to match context entities involved in steps. This is also a non-invasive approach as the joinpoints are not specified in the basic use cases.

The advices contain steps and extensions as found in use cases. Extensions are elements of a use case which specifies what a use case does when it cannot complete the main scenario. The advice specifies in an AspectJ like manner whether it is executed before, after or around rsp. instead of the specified step. If the advice contains an extension the aspect can specifiy whether it wants to abort the rest of the use case.

[1]With ≪extend≫ the *extension points* have to be explicitly listed in the extended use case.

2.2 Our Approach

For dealing with aspects which have internal structure and complex behaviour we advocate the approach by [16] which encapsulates these aspects as use cases. Moreover, we see the language ObjectTeams/Java [7] as a suitable target platform for internally structured aspects because it supports collaboration of roles. Thereby, the transformation between requirements, analysis, and design can be made more seamless.

Our approach aims at covering the development lifecycle including the following phases:

- requirements modeling
- analysis
- design
- implementation

We shortly sketch the intention of each of these phases.

Requirements modeling

Requirements modeling is concerned with providing *rigorous* use case models [5]. Such models consists of graphical use case diagrams and of detailed descriptions of each use case containing main and secondary scenarios in terms of steps, as well as triggers, pre- and postconditions. An activity diagram can be used to specify a use case and its steps more formally.

In this contribution we will base our development process on the aspect-oriented extension to use cases by [16]. Their approach can be seen as a refinement of standard use case driven modeling. In general, a development process targeting ObjectTeams/Java as a design and implementation platform can start from a refined use case modeling as well as from a standard use case modeling. While the refined modeling already makes some criteria explicit which will steer the subsequent analysis and design process these criteria can also be applied starting from a standard use case modeling approach.

In this contribution we will not consider non-functional requirements (NFRs). Also, we will not consider viewpoint driven requirements modeling, although this might fit very well with our approch.

Analysis

We view our analysis step in correspondence with standard *object-oriented analysis* such as in [10]. This step takes as input the domain model and the sequence-oriented activity diagrams rsp. the use case steps and has to identify collaboration using nested method calls among the domain entities. In addition, new objects are identified for the user interface at the system boundary or for controling the behaviour of a use case. The analysis results mainly in providing scenarios for the identified collaborations by means of UML interaction diagrams which we will use in the form of sequence diagrams here.

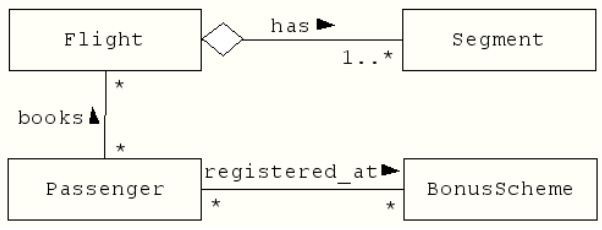

Figure 1: Flight booking domain class diagram

We also use the standard UML stereotypes from the profile for software development processes [5] for classifying the domain objects and the newly introduced objects. A ≪boundary≫ object represents the system boundary and handles user input. A ≪controler≫ object steers the control flow within a use case. An ≪entity≫ object represent passive objects participating in a number of collaborations (see figure 6 for the graphical notations of the stereotypes.)

We perform this kind of analysis also for the crosscutting use cases. Note, that sequence diagrams cannot describe complete class behaviour. We omit showing how new objects and methods are reflected in augmenting the class diagram. Note, that the analysis step is independent from our target language ObjectTeams/Java and hence does not distinguish between classes, roles, and teams.

Design

The design targets ObjectTeams/Java [7]. The design takes the previously developed analysis models as input and aims at identifying classes, roles, and teams. Objects from different use cases rsp. their analysis sequence diagrams are merged into classes or become roles. Collaboration among objects in these sequences is correspondingly transformed into class-role interaction.

To capture designs for this target we have developed a UML profile called UFA based on class diagrams [6]. UFA captures all relevant concepts of ObjectTeams/Java needed during the design, i.e., the concepts of teams and roles. Sequence diagrams are extended to show role-based behaviour [3].

Implementation

Designs in UFA accompagnied with extended sequence diagrams can easily be implemented in the ObjectTeams/Java programming language. Here, we cannot go into details. Implementations in ObjectTeams/Java have been presented in [7, 18].

3. ELICITATING AND MAPPING ASPECTS

In this section we will demonstrate how to identify crosscutting use cases with internal structure and behaviour and how to map them to a design for ObjectTeams/Java. We will illustrate our concepts with a running example which is a flight booking system.

Flight booking example

Passengers book flights consisting of individual segments. Passengers are registered in the system and can be unregistered as well. Both is conducted by staff members on behalf of passengers. Flights and flight segments can be administered by staff members. Unbooking of flights is possible, too.

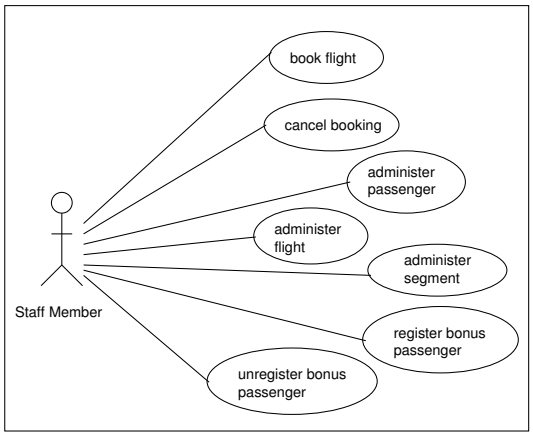

Figure 2: Flight booking use cases

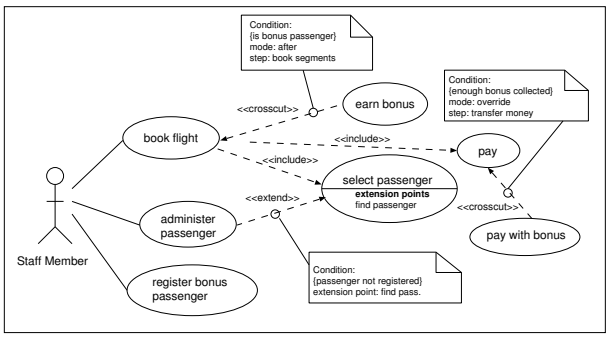

Figure 3: Refined flight booking use cases

Passengers can register for a bonus scheme which enables them to collect bonus miles for flights they book. When a flight is booked the bonus is collected depending on the length of the flight segments and possibly depending on other conditions (for example the number of segments of the flight).

3.1 Requirements Modeling

Figure 1 shows a simple initial class diagram for the flight booking domain. The passenger books flights which consist of several segments. A passenger can be registered for a bonus scheme. Note, that this diagram specifies the structure of domain entities and that it is developed in parallel with the initial specification of system behaviour by means of use cases.

Figure 2 shows an initial use case diagram identifying the main tasks of a staff member. Figure 3 shows how a subset of these use cases, which are related to use case *book flight*, is refined with additional use cases and with use case relationships. We also provide a detailed description of the use cases which will be of interest in the following. The use case *book flight* includes the use case *select passenger* which is ex-

tended by the use case *administer passenger* meaning that the included use case is mandatory while the extension is only needed if the passenger's data is not yet in the system. The use case *book flight* also includes the use case *pay*.

The steps of use case *book flight* are described in table 1. The precondition of this use case is a client wishing to make a booking. The steps are also presented in the activity diagram in figure 4.

Use case	book flight
Actor	staff member
Trigger	passenger gives booking order to staff member
Precondition	flight exists
Postcondition	each segment of the flight is booked
Main scenario	1. select flight 2. select passenger 3. reserve segments 4. book segments 5. pay

Table 1: Use case book flight

Our focus lies on the bonus requirements. In refining the *book flight* use case we have defined a use case *earn bonus*. This use case describes how a registered bonus passenger is able to receive a bonus for a booked flight. We consider this a crosscutting use case following [16] because the bonus collection is not an essential part of the booking and does not contribute to the goal of booking either. This fulfills the first criterion for crosscutting relationships as outlined in 2.1. In our small scale example we cannot discuss the criterion whether this use case also contributes to other use cases. But it is quite common today that the same bonus system does not apply to booking flights only but also to purchasing other goods.

Use case	earn bonus
Actor	staff member
Trigger	purchase of bonus segments
Precondition	passenger registered
Postcondition	bonus for each segment earned
Main scenario	1. get segment distance 2. calculate bonus for segments 3. sum 4. credit

Table 2: Use case earn bonus

Technically, we use the stereotype ≪crosscut≫ to denote the special relationship between the two use cases. The condition under which the use case *earn bonus* is executed is denoted as {*is bonus passenger*} saying that the passenger has to be registered for the bonus in order to collect a bonus. The step after which the use case can execute is *book segments*. The mode is *after* saying that the bonus is collected after the booking is completed. The steps of the use case *earn bonus* are presented in table 2 and in figure 5. The trigger of this use case is not a user interaction such as in the use case *book flight* but an activity in another use case which is suitable for earning a bonus.

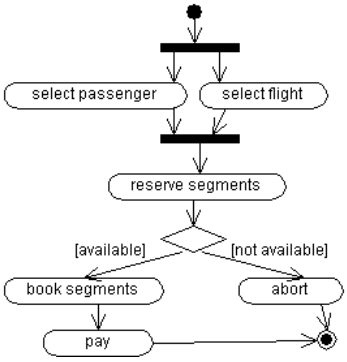

Figure 4: Activity diagram for book flight

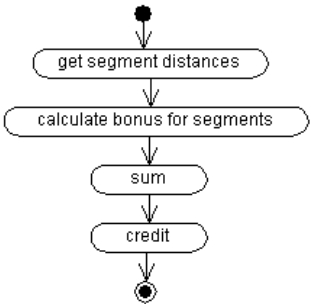

Figure 5: Activity diagram for earn bonus

The use case *pay with bonus* is also considered to be crosscutting assuming that for a bonus passenger it is checked automatically whether he or she can pay with the bonus and assuming that the same could be applied to paying other goods. This example is not discussed further in this contribution.

3.2 Analysis

As explained in the approach, this is a highly creative phase which requires to translate from the sequence-oriented activities and steps into the message rsp. method-oriented object collaborations with nesting of calls.

We show such a translation for the use case *book flight* from figure 3. This translation involves the domain classes (see figure 1) for identifying objects involved in a use case. It involves also the introduction of new objects. In the example, we identify a boundary object *bookingUI* for interaction with the staff (see figure 6). The controler object *bookhandler* controls the booking by sending corresponding messages to the entity objects *passenger*, *flight*, and *segments* (encapsulating a multiobject) The latter three correspond to the domain classes.

Typically, not all steps (see table 1) rsp. activities (see figure 4) can be mapped one to one to a method call. For instance, the activity *book segments* cannot be mapped to a method call from passenger to segments because there is no link from passenger to the segments (see class diagram in figure 1). Therefore, the call has to be mediated for example by the flight object.

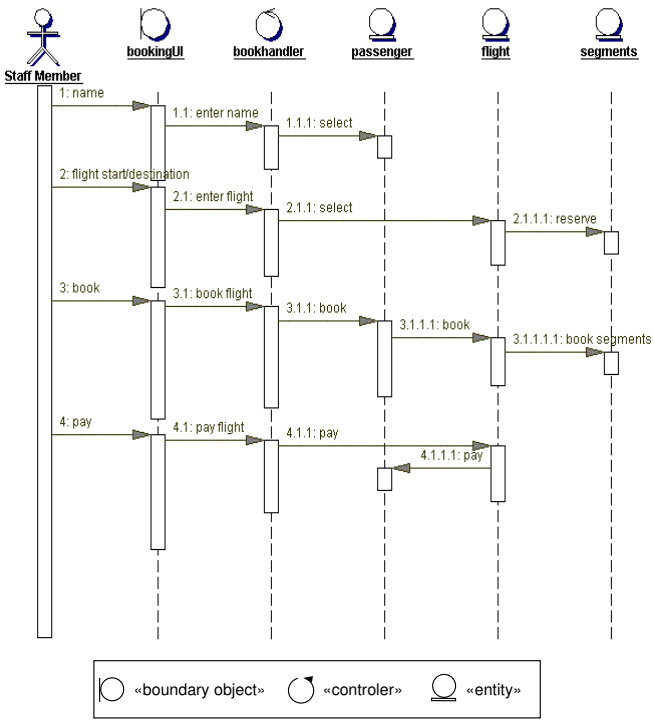

Figure 6: Analysis sequence diagram for book flight

We also envision such a translation for the crosscutting use cases *earn bonus* from figure 3 resulting in the sequence diagram at figure 7. Apparently, this use case is not triggered by a staff member interaction. Therefore, we invent a method call *earn flight bonus* as a trigger without saying at this point in time how this relates to the step *book segments* chosen as a trigger in the use case diagram. We will consider this only in the design phase. Apart from this the sequence diagram is established as usual. It contains a controler object *bonushandler* and two entity objects *passenger* and *segments*. Note that in this step we do not distinguish between a passenger involved in booking and a passenger involved in claiming a bonus although it is obvious that different parts of their behaviour participate in the two different sequence diagrams. The sequence diagrams are different views of the same domain entity.

3.3 Design

UFA and extended sequence diagrams

UML for aspects (UFA) is an extension to UML based on the concepts of the programming language ObjectTeams/-Java [6]. UFA captures the programming model behind ObjectTeams/Java and allows a straight forward mapping into ObjectTeams/Java.

The basic idea of this programming model is to decorate classes with *roles* which are special classes. A decorated class is called *base*. An object and its role instance can be seen as an aggregated entity. A role instance can interact with its decorated object in two ways: It can intercept method calls to the decorated object. This is called a *callin*. Callins can be of type *before*, *after*, or *replace*. A role can call

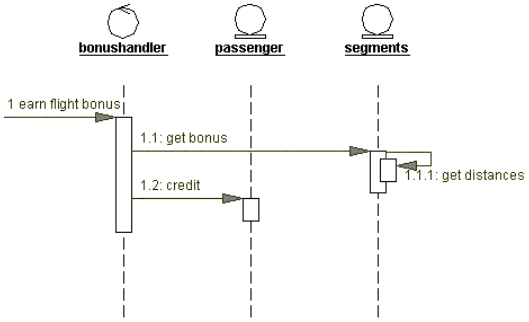

Figure 7: Analysis sequence diagram for earn bonus

methods of the decorated object without specifying the object. This is called a *callout*. These two kind of interactions are also referred to as *method bindings*.

Teams are a structuring concept to encapsulate collaborating roles. Teams have class features. A base object can have only one role instance per role class and per team instance.

A callin, i.e., a method call interception, can be compared to an advice weave as found in Aspect-Oriented Programming [9]. Roles and their enclosing teams thereby can be used to encapsulate crosscutting behaviour.

ObjectTeams/Java provides fine-grained programmatic control over activation of teams which controls the behaviour of its role instances with regard to callins. A team instance can be either

- *ACTIVE*, meaning that callins are executed. A callin will also implicitly create a role instance if it didn't exist yet.
- *INACTIVE*, meaning that no callin will be executed.
- *FROZEN*, meaning that only callins for existing roles are executed. No roles are implicitly created.

Independent from the activation, a role can always be created explicitly using team-level methods with a special *base-as-role* signature containing the base class and the role class.

UFA-diagrams capture roles, teams, decoration of base classes, and the method binding between roles and base classes (see figure 8).

- Although teams share the standard behaviour of a class, they are denoted as a ≪team≫ stereotype of a UML package because they contain role classes.
- If the roles of a team decorate base classes, the base classes have to be encapsulated by a package. The team is said to adapt the base package which is denoted by a stereotyped dependency ≪adapt≫ between the two packages.
- A role is presented using a UML class with a few extensions. If the role is bound, the role name is followed by = and the decorated base class name.

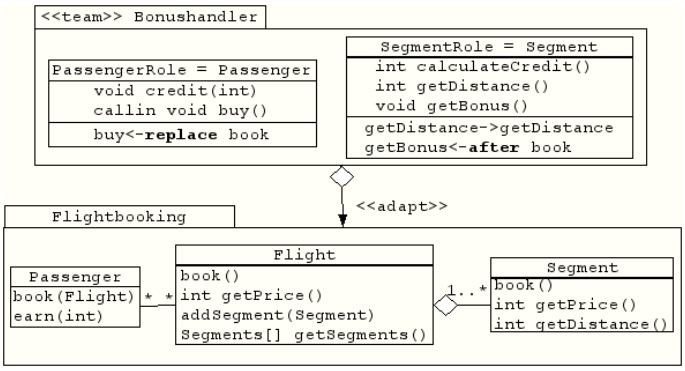

Figure 8: UFA diagram

- Method bindings are presented in a separate compartment of roles. There are two kind of bindings. Call-ins intercept base method calls and map them to role methods denoted by a left-pointing arrow. Callouts are role methods which are mapped to base class methods denoted by a right-pointing arrow.

Our sequence diagram notation takes up the idea to integrate aspects into the main flow of control such as in [2] which depict that a method call is intercepted by drawing a circle with a cross on a message arrow in the sequence diagram. This circle is linked to an icon which stands for the aspect entity.

We have adapted the part of the notation for the method call interception (see figure 9) but omit the link to the aspect entity. Instead, our notation goes further by showing the behaviour exhibited on behalf of the aspect rsp. its entities [3]. Adapting this to ObjectTeams/Java means that we show additional messages interchanged among roles and teams. To this end we have introduced a new stereotype ≪role≫ object, denoted graphically as a dice with a shadow (see figure 9). Teams are depicted as a grey box enclosing their role instances. Activation bars of role instances are shaded in order to indicate that this control flow is executed within a team.

Design Mapping

To achieve seamlessness it is desirable to keep the use case modularity also in the design. As already pointed out, here we deal with use cases which (i) have been identified as crosscutting use cases, and (ii) which have been refined by an object-oriented collaboration with complex structure and behaviour.

A more detailed analysis of such collaborations results in the following characteristics. Note, that not all of them will always be true for each crosscutting use case refined by a complex collaboration.

1. A crosscutting use case is refined by a collaboration with a number of different objects and different relationships between them.
2. A crosscutting use case needs context information to operate.
3. The relationship between objects of the collaboration refining the crosscutting use case differs from the relationship of the same objects in the collaboration refining the base use case.
4. The objects from the collaboration refining the crosscutting use case are from a different domain than the objects from the base use case but they share certain attributes.

The first characteristic asserts that the use case has a reasonable complex refinement by means of a collaboration. In a design, it is desirable to encapsulate this collaboration. The other three characteristics describes the interface of this collaboration to its context, more precisely to other objects. In a design, it is desirable to provide control over this interface.

A team supports this kind of encapsulation while at the same providing a controllable interface to part of its environment, i.e., the base classes. Each of the above characteristics is supported by concepts of a team and its roles.

1. Complex structure is supported by role attributes and by associations between roles.
2. Role instances can obtain information about the decorated objects using callouts.
3. Roles can have associations to roles without corresponding associations on the base class level.
4. Roles and their base instances are conceptually the same entity, thereby roles share the attributes of the base.

Therefore we conclude that a team is an ideal candidate for mapping a crosscutting use case. This becomes our main design principle. Note, that in order to apply this principle and to map a crosscutting use case to a team, not all of the four above characteristics have to be fulfilled.

In the following we apply our main design principle to the flight booking example. Note, that in the context of this paper we only consider a design for a single-threaded environment. ObjectTeams can however also be used in a multi-threaded context.

The use case *earn bonus* is considered a candidate for a team. In addition to being identified as crosscutting, it has behaviour and a different view on the domain entities than, for instance, the use case *book flight*.

From the analysis we know that the two entities *passenger* and *segment* appear in two different collaborations. They exhibit different behaviour in each analysis sequence diagram. However, it is obvious that the passenger booking a flight and the passenger earning the bonus share certain properties such as their identity. Such an overlap in structure can be molded into roles decorating classes. The resulting roles *PassengerRole* and *SegmentRole* are depicted in figure 8. The controler object *bonushandler* becomes a

team. Its main task will be to take control over the instantiation while controling the collaboration itself will be implemented with callins as explained in the following.

From the use case diagram we know that the behaviour of use case *earn bonus* is executed after step *book segments* in the use case *book flight*. This is a hint to a callin which has to be expressed in terms of the method calls from the analysis sequence diagram.

It is fairly straight forward to determine the method call *book segments* as the candidate for a callin because it is the last method call involved in booking. The mode *after* can also be kept. The behaviour to be inserted is described in the analysis of use case *earn bonus*. The method *get bonus* has to be executed as the callin on the role decorating the segment The call *earn flight bonus* thereafter becomes redundant. This is depicted in figure 9 with the the circle with the cross and the branching of the call *book segments*. Subsequently, the segment role has to determine the distance by accessing its base class segments. Note, that the stars on the method calls denote that these calls are executed many times because the target is a multiobject.

After calculating the bonus the segment has to credit the bonus to the passenger which is not known to a segment in the analysis of use case *book flight*. However, it was assumed to be known in the analysis of *earn bonus*. We can use a second callin to make the passenger available as a role in the context of the segment role so that the two can collaborate. When the method *book* is called on a passenger we make a callin to the passenger role method *buy*. This method stores the passenger role as a reference inside the team bonushandler so that subsequent calls can use it. Method *buy* is not only a callin but replaces the original method book. However, it calls the original method inside *buy* similar to *proceed* in an around advice in AspectJ. Each replace callin to *buy* will change the reference to the passenger role inside the team. After this preparation, the segment role can call *credit* on the passenger role. After the base method call, the replace call in will set the reference to null at the end. The circle with the dot denotes such a replace callin (see figure 9).

This approach requires that the corresponding team instance is activated. In order to control that only bonus passenger will have a role instance, the following steps have to be taken: the team has to be FROZEN and bonus passenger roles are created using a team level method public void register(Passenger as PassengerRole){//empty}. The replace callin sets the team activation to ACTIVE, calls the base method, and after this, sets the team activation to FROZEN again. During the team activation, the call to book on a segment can trigger the after callin.

Note, that the role rsp. team behaviour is not only triggered by a single method call from the base, but by two subsequent calls. The team will only be active for a specific kind of control flow happening in the base. This is not surprising because we set out to map use cases with complex internal behaviour to aspect-oriented programming. This may also include the case that the control flow in a crosscutting use case depends on the control flow of a base use case.

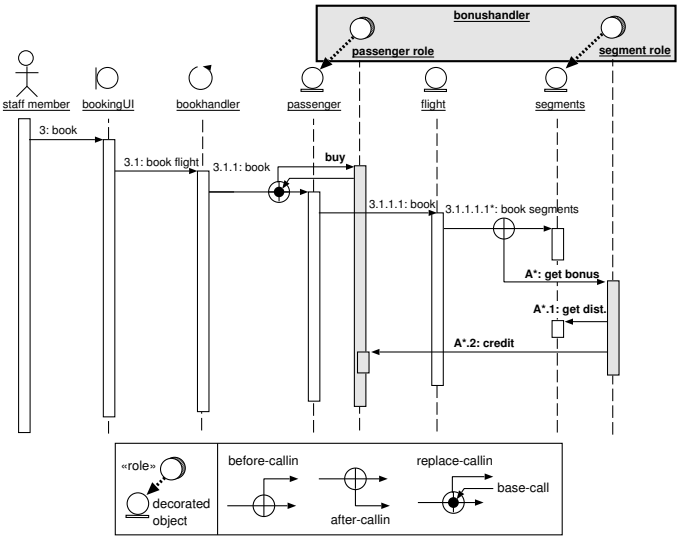

Figure 9: Sequence diagram with roles

Figure 9 demonstrate the complete behaviour from the use cases *book flight* and *earn bonus*. In the final UFA diagram we also show the Java base classes (see figure 8) which are refinements of the domain classes.

Having finished the UFA and sequence diagram we point out once more that it is straight forward to derive an implementation in ObjectTeams/Java from these models. An example of the flightbooking implementation can be found in [18].

4. RELATED WORK

Jacobson [8] argues that aspect-oriented programming provides the missing link in order to keep the use case modularity in the design and implementation. He proposes to map extension use cases to aspects while other use cases will still be mapped to standard object-oriented components. An extension use case refers to extension points which have to be inserted into the sequence of actions of the extended use case. In contrast, the crosscut relationship does not require to prepare the extended use case for the extension.

Recently, the first Model Driven Architecture (MDA) approaches have taken up ideas from AOSD. One specific approach called executable UML [14] proposes to strictly separate concerns in so called domains and to specify automated integration which makes extensively use of AOSD concepts such as joinpoints, method call interception and weaving. However, that approach only coarsely sketches ideas which have not been instantiated yet. It is comparable to our goals in that it envisions weaving for richly structured aspects.

In a similar respect, the process proposed with the language AspectU (for use case level aspects) envisions partial generation of subsequent models from AspectU [15].

Theme/UML [4] provides a notation for parameterising packages over methods. Sequence diagrams contained in such a package can make use of these parameters as triggers of sequences. These diagrams go one step further than our anal-

ysis sequence diagrams for crosscutting use cases because they already show that these sequences will be triggered by method calls and are different from other analysis sequence diagrams. When Theme/UML packages are bound to other packages, the parameter methods can be bound to actual methods meaning that the actual methods can be intercepted. Here, our extended sequence diagram notation goes one step further in specifying the integrated sequence.

Role-based modeling has a long tradition. We have molded these ideas into the programming language ObjectTeams/Java while combining them with concepts from Aspect-Oriented Programming such as method call interception.

5. CONCLUSION AND OUTLOOK

We have described concepts for a process with which we cannot only elicit but also consistently refine aspects with an internal structure and complex behaviour. We have demonstrated with an example how the transformation from a standard object-oriented analysis to a design for Object-Teams/Java can be guided by design principles. The transformation of analysis collaborations into collaborations with roles and classes results from applying these principles and documents the design apart from the purely structural UFA diagrams.

While the use case model by [16] provides an advantage over standard use case modeling we feel that the relationships between ordinary and crosscutting use cases are even more manifold than what is currently captured. In the example we gave, it turned out that more than one method call interception is needed to specify how the two collaborations from the two different use cases are intertwined. Our main point is that using isolated steps as single triggers fall short of identifying dependencies between complex behaviour. Also data dependencies should be dealt with.

In the future, we also plan to integrate view points into the proposed process. We believe that this technique may be another source for deriving richly structured aspects.

6. REFERENCES

[1] *Proc. of First International Conference on Aspect Oriented Software Development*, Enschede, Netherlands, 2002. ACM Press.

[2] M. Basch and A. Sanchez. Incorporating Aspects into the UML. In *Proc. Workshop on Aspect Oriented Modeling at AOSD 2003*, 2003.

[3] S. Binner. Erweiterung eines UML-Werkzeuges um aspektorientierte Modellierung für Object Teams (german). Master's thesis, Technical University Berlin, 2004 (in preparation).

[4] S. Clarke and R. Walker. Towards a standard design language for AOSD. In AOSD'02 [1], pages 113–119.

[5] Object Management Group. Unified Modeling Language. `http://www.omg.org`.

[6] S. Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at First International Conference on Aspect Oriented Software Development (AOSD)*, 2002.

[7] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Proc. Net Object Days 2002*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2002.

[8] I. Jacobson. Use Cases and Aspects - Working seamlessly together. *Journal of Object Technology*, 2(4):7–28, 2003.

[9] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.

[10] P. Kruchten. *The Rational Unified Process*. Addison Wesley, 2000.

[11] Object Teams home page. `http://www.ObjectTeams.org`.

[12] A. Rashid, A. Moreira, and J. Araujo. Modularisation and composition of aspectual requirements. In AOSD'02 [1], pages 11–20.

[13] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *Proc. IEEE Joint International Conference on Requirements Engineering*, pages 199–202. IEEE Computer Society Press, 2002.

[14] M. Balcer S. Mellor. *Executable UML*. Addison Wesley, 2004.

[15] J. Sillito, C. Dutchyn, A. Eisenberg, and K. DeVolder. Use case level pointcuts. In *Proc. ECOOP 2004*, Oslo, Norway, June 2004.

[16] G. Sousa, S. Soares, P. Borba, and J. Castro. Separation of crosscutting concerns from requirements to design: Adapting the use case driven approach. In *Proc. Early Aspects Workshop at AOSD 2004*, 2004.

[17] TOPPrax home page. `http://www.topprax.de`.

[18] M. Veit and S. Herrmann. Model-View-Controller and Object Teams: A perfect match of paradigms. In *Proc. AOSD'03*, Boston, USA, March 2003. ACM Press.