JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007 2007

Improving Alignment of Crosscutting Features with Code in Product Line Engineering

Christine Hundt, Technische Universität Berlin, Germany
Katharina Mehner, Siemens AG, Germany
Carsten Pfeiffer, Dehla Sokenou, GEBIT Solutions, Germany

Feature models used in product line engineering often include features that crosscut other features. These features cannot be cleanly modularized using object-oriented techniques and are the source of scattering and tangling in implementation modules. This significantly complicates the traceability of features during the development and maintenance of a product line and during the instantiation of a product.
This paper proposes a model-driven approach for mapping features to a design in the aspect-oriented role-based language Object Teams. The approach has been evaluated in an industrial case study for developing a security product line that can be applied to several applications using aspect bindings.

1 MOTIVATION

One of the challenges of today's software development is the management of variability. While a standard software system may be inappropriate for many customers, individual products become too expensive. Being able to provide different variants of a software product reduces costs and means a significant advantage on the market. Software product lines are an approach to conquer the demands of variable products. A software product line is a set of products with common characteristics that is developed from a common set of core assets.

Feature models are a means to capture commonalities and variabilities of a product line. They often include features that crosscut other features. Using standard object-oriented techniques, these features cannot cleanly be modularized. A crosscutting feature would affect more than one implementation module (*scattering*), and an implementation module would realize different features (*tangling*). For variable features, i.e. optional or exclusive features, sometimes a separate implementation for each feasible combination is needed. This mis-alignment of features with implementation modules significantly complicates the traceability of features during the development and maintenance of a product line and during the instantiation of a product. Generative techniques try to overcome this situation by generating variable code into code fragments. The scattering and tangling still exists in the generated code and the evolution of partly generated software is very difficult to manage.

In this paper, we demonstrate how the aspect-oriented role-based programming

Cite this document as follows: Christine Hundt, Katharina Mehner, Carsten Pfeiffer, Dehla Sokenou: *Improving Alignment of Crosscutting Features with Code in Product Line Engineering*, in Journal of Object Technology, vol. 6, no. 9, Special Issue: TOOLS EUROPE 2007 2007, pages 417–436,
http://www.jot.fm/issues/issues_2007_10/paper21

language Object Teams [8, 9] improves the alignment of features with code in the implementation of a product line and in the instantiation of a product. Although the Object Teams programming model was not designed for feature-oriented programming and product line engineering it proved to be quite suitable for that. During *domain engineering*, a software product infrastructure is built using a model-driven approach for mapping a feature model to a design in Object Teams. In most cases, a one-to-one correspondence between features and Object Teams implementation modules can be realized. During *application engineering* a product instance is derived using Object Teams mechanisms. Our approach also supports the implementation of a product line that is itself an aspect. Object Teams supports binding the product instances as an aspect to a base application. Additionally, we propose guidelines for providing flexible binding mechanisms in the case that product line instances are to be bound as aspects to different applications. The approach has been applied in an industrial case study to a security product line in Object Teams as part of the practical evaluation project TOPPrax [23].

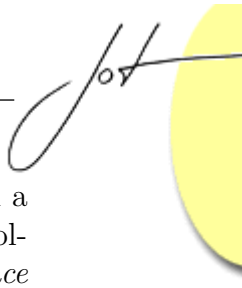
The paper is structured as follows. A brief introduction to feature-oriented programming with Object Teams is given in Section 2. Section 3 presents rules for mapping feature models to an Object Teams design and applies them to derive the security product line infrastructure. Section 4 describes the instantiation principles using Object Teams and their application to the security case study. The advantages of our approach are discussed in Section 5 and its relations to other work in Section 6. Section 7 concludes and gives an outlook.

2 FEATURE-ORIENTED PROGRAMMING WITH OBJECT TEAMS

When establishing a product line, a feature model is often used during domain engineering to capture commonalities and variabilities. The features drive the development and maintenance of a product line and the product instantiation. Hence, a seamless traceability of features through alignment of features with implementation modules is desirable. Variable features, i.e. optional or exclusive features, need to be mapped to implementation modules that can be selected individually during product instantiation. Seamless traceability also provides opportunities for a model-driven development that has the potential for automation.

Our underlying notion of a feature is a characteristic or trait of a software system that describes a coarse-grained requirement. In the context of product lines, a feature is either common to all product line instances or serves to distinguish different instances of a product line. A feature often describes something visible to the user but it can also describe an implicit technical requirement.

A product line can be modeled using the feature model notation of [6]. The feature model defines *mandatory* features (denoted as a filled dot above the feature node) and *optional* features (denoted as an empty dot) as well as *alternatives* (denoted as an arc between features). The notation supports the constraints *exclusion*,



inclusion and *influence* relationships between features. Features are organized in a hierarchy. Our notion of the relationships between features in such a hierarchy follows the commonly used semantics, i.e. *part-of* and *is-a* hierarchies. The *influence* relationship can be considered as a *crosscutting* relationship to other features of the same level and of sublevels.

Crosscutting features and variable features have been identified as the difficult cases regarding alignment with implementation modules. The behavior of a crosscutting feature affects many other modules and these modules will become tangled with the behavior of crosscutting features. For integrating the behavior of variable features, mandatory modules will have to provide explicit hooks. For both cases, it is desirable to modularize the behavior and to integrate it with the other behavior without affecting the existing modules. Object-oriented modules are not sufficient to capture features that crosscut other features. Their support for variable features is limited to class-based inheritance.

The observation that object-oriented implementations of crosscutting requirements or features lead to tangling and scattering has been the origin of aspect-oriented programming [11]. In order to provide a scalable approach, not only aspect-oriented mechanisms for behavior interception at joinpoints are needed. A feature is often not a singular behavior but adapts a collaboration of classes in an object-oriented implementation of features. Therefore, support for modules that capture a collaboration of crosscutting code and support for inheritance of collaborations is required. This idea is pursued by aspect-oriented role-based languages like Object Teams [8].

The Object Teams Programming Model

Object Teams [8, 9] is a modern programming model advancing object-orientation with new modularization concepts. A full-featured development environment for Object Teams (OTDT) is available [16] as a set of Eclipse plug-ins, including an incremental compiler, code completion, structure views, a debugger, and more. As Object Teams has evolved from Aspectual Components [13], it combines valuable properties of the worlds of aspects and components. It introduces a new module concept, the *team*. A team is a package that groups classes. At the same time, a team is a class and therefore supports e.g. inheritance and instantiation. A team instance is a container for objects defined within and provides a context for their collaboration. The classes contained in a team are called *roles*, and they can decorate other classes. A role is bound to a class by declaring a *playedBy*-relation. A decorated class is called *base*. A base object can be decorated by zero or one role instance per role class and per team instance. A role can interact with its base object in two ways:

- A role class can specify *callin* method bindings. Thereby, a role instance intercepts method calls to its base object in order to execute a role method (instead

or additionally). In aspect-oriented programming, the elements referred to in callin bindings are called *join points*[1] [10].

- A role class can specify *callout* method bindings, allowing role instances to forward method calls to base members (methods or fields).

Bindings can access any methods and attributes of the base objects including private class members. Roles and their enclosing teams encapsulate behavior that can adapt a collaboration of base classes in a specific context through the means of bindings. The team concept is scalable in two respects: Firstly, a role of a team can itself be a team. Secondly, the base class to which a role is bound can itself be a team or even a role class of another team. Although there is no counterpart in the language it is useful to use an *adapts*-relation to denote the relationship between a team and the elements it adapts.

The adaptation capability of a team can be controlled at runtime. A team instance can be *activated* or *deactivated*. If a team is active, all callin bindings are enabled; if it is deactivated, they are effectless. Deactivating a team instance does not affect its state nor the state of its contained roles. This state persists throughout the life-time of the team instance. Besides activation at runtime, a team can also be activated at program start time using a *launch configuration*. This implies that a single instance of the team is created.

A team can be declared as abstract. An abstract team may contain abstract roles that can be bound to base classes in a concrete subteam class. Roles in a concrete team implicitly inherit from corresponding roles of the super team, where correspondence is established by name equality between role classes. The distinction between definition of roles and binding of roles to base classes is also used from a methodological perspective to separate logic from binding and to make binding a declarative programming paradigm. This essence is captured in the *Connector* pattern. A *subteam* becomes a connector if it only declares callin and callout bindings. Moreover, the binding step can be accomplished using a visual drag and drop editor in the Object Teams development tooling (OTDT).

The modularization concepts of Object Teams (teams, roles) constitute units capable of realizing the functionality of features, allowing for a better alignment of features with implementation modules. The following sections detail the mapping of features to Object Teams.

3 DOMAIN ENGINEERING WITH OBJECT TEAMS

Our domain engineering approach is driven by feature models from which an Object Teams design of a product line infrastructure is derived. The aim is to align features

[1] Method execution join points must be referred to explicitly in Object Teams. Even though this is sufficient for our approach, a richer join point language, comparable to the one implemented in AspectJ is in preparation.

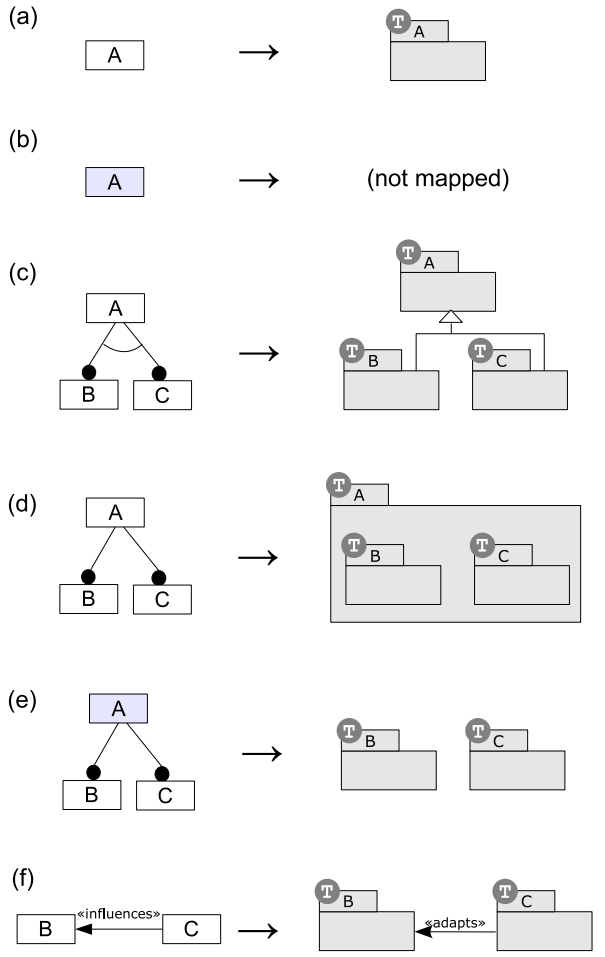


Figure 1: Mapping feature model elements to Object Teams

with modules of a programming model. We present in detail the mapping of features to a static design in Object Teams. After the mapping, the behavior is implemented.

Mapping Features to Object Teams

Ideally, one feature should correspond to one module; in Object Teams terms: a *team*. A team is the ideal unit to be selectively included in a program. This can either be a compile-time decision or a launch time decision through team activation.

The mapping we present here is driven by the distinction between mandatory and optional features and by the distinction between part-of, is-a and crosscutting semantics of features. These mapping rules derive a static structure. In addition, the feature model is mapped to Object Teams constructs that control the activation of teams and roles at runtime. Provided that the semantic distinction between part-of, is-a and crosscutting can be derived from a feature model, the static part of the design can even be automatically generated using the following principles.

Structural Mapping

We define the following mapping (see Figure 1):

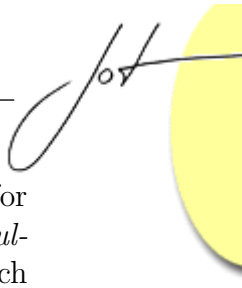
- Basically, each feature is mapped to a team (a). If a feature should not be mapped, it has to be explicitly excluded from the mapping (b). This means that the feature is fully described by its subfeatures and defines no additional behavior.
- Alternative features, mandatory or optional, are considered an *is-a* relation to their super feature. They are mapped to teams that inherit from the team corresponding to the super feature (c). In this case, subfeatures are considered as extensions of their super feature.
- Non-alternative subfeatures are considered as part of their super feature. A mandatory, optional or exclusive feature that is in a *part-of* relation to a super feature is mapped to a team nested inside the team that corresponds to the super feature (d).

  If the super feature is marked as unmapped, the whole is lacking in the whole-part relationship. In this case, a different mapping is used. Subfeatures are mapped to independent teams (e).
- An *influences*-relationship between a feature and another feature has crosscutting nature and is mapped to an adaptation between teams. The influencing team has to adapt the team which it influences. If a feature has more than one influences-relationship, the corresponding team has to adapt all teams which it influences. The adaptation is realized by class (*playedBy*) and method (*callin*, *callout*) bindings.

More fine-grained features can be mapped to roles of a team. Each of the roles is introduced on the level of the given architecture and will be bound later. At present, the internal structure must be implemented, e.g. using the role creation wizard of the Object Teams Development Tooling. In the future, the internal structure should be part of the feature model. Such an example will be discussed on a concrete level in the section where we present the security product line example. Here, adapter roles can be generated for subfeatures that describe a choice between different GUI elements (see Section 4).

Controlling Selection, Activation and Instantiation

As stated above, mandatory, optional and alternative features are statically mapped to structural elements in a one-to-one correspondence. The selection of features to be contained in a concrete product line instance is done dynamically via instantiation and activation of teams. Constraints (includes, excludes) should be enforced by the tool used for product line instantiation.



In addition, there may be constraints depending on dynamic conditions. If, for example, a constraint only states that two features should never be *active simultaneously* within the same application, this can not be ensured statically. Such constraints can be stated and enforced at run-time using guard predicates [17], a feature of Object Teams which allows to further fine-tune a team's activation at runtime. Such constraints might be additionally annotated in the feature model.

In general, we propose to have a team that encapsulates the activation of teams. As a consequence, only one team has to be activated (statically) via a launch configuration. In the following, we call such a team an *activation team*.

Security as Product Line Aspect

In an object-oriented security implementation, one will typically find crosscutting code. Authentication is tangled with authorization if the latter is present, and in the implementation of authentication, login mechanisms and logout mechanisms are tangled. This makes security an interesting candidate for applying our approach.

In this section, we describe the application of the mapping rules in an industrial case study in which we have realized a product line for a security aspect. Security was developed as a product line in order to accomodate the need for different security requirements. Authentication should be supported and optionally authorization. Both should be supported using different options, e.g. different kinds of logins and different authorization models. The security implementation uses the *Java Authentication and Authorization Service* (JAAS). Security was developed as an aspect because it should be possible to non-invasively add it to existing applications, i.e. without changing them. This also facilitates independence from the technology used for their implementation, i.e. object-oriented or even aspect-oriented technologies. Moreover, the aspect solution supports different ways of binding it to a given application to increase re-use and flexibility.

Feature Model of the Security Component

The security component is intended to provide *access control* by user *authentication* and *authorization*. A user management stores login names and passwords. For authorization, it is assumed that access restrictions for each user can be specified. Authentication and authorization are the functional requirements detailed in the following. Authentication is the minimal function required for access control. Authorization is dependent on authentication. In the feature model (see Figure 2), this is expressed by modeling authentication as a *mandatory* feature (filled dot above feature node) while authorization is an *optional* feature (empty dot). If only authentication is requested, the authenticated user has complete access to the application.

The minimal requirement for authentication is the *login* feature. Login is carried out either through a *user interface* or through *operating system user identification*,

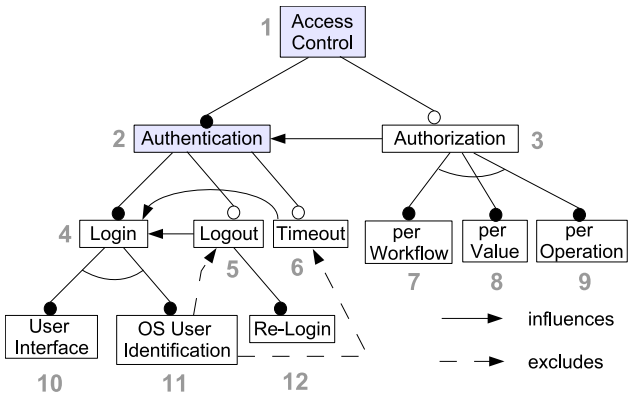


Figure 2: Feature Model

in the feature model represented by the arc between the corresponding features.

An optional *logout* feature supports to explicitly log out from the system. In the simplest variant, logout is implicit in the shutdown of the application. After logout, a new user or the same must be able to *re-login*. Optionally, a *timeout* is available to logout a user automatically after a configurable time span has elapsed without user interaction.

The influences-relationship between the two options and the login feature has the following characteristics. Logout and timeout both require the login feature and influence its behavior. If selected, they can assume that the mandatory login feature is present. Logout and timeout are not supported if login is realized by operating system user identification. This constraint is realized by the *excludes* edge from "OS User identification" to logout and timeout respectively.

Authorization is supposed to cover business objects containing data (*per value*) but also workflows (*per workflow*) or certain operations (*per operation*). Different access rights and different user roles are distinguished.

In the following, we concentrate on the authentication feature which in our case is the more interesting feature considering the options and the binding to the base application. Authorization is also implementated in the product line and is configured via a configuration file which defines users and permissions in a usual way, so that only a minimal set of bindings to the base system is needed.

Deriving the Architecture of the Security Component

The resulting architecture of our example – the security component – is shown in Figure 3. This architecture skeleton can be directly generated from the feature model. Comparing the given architecture with the feature model from Figure 2, we see that the teams are derived directly from the feature model. The feature and the corresponding team are marked with the same number.

The features *access control* and *authentication* are not mapped, according to

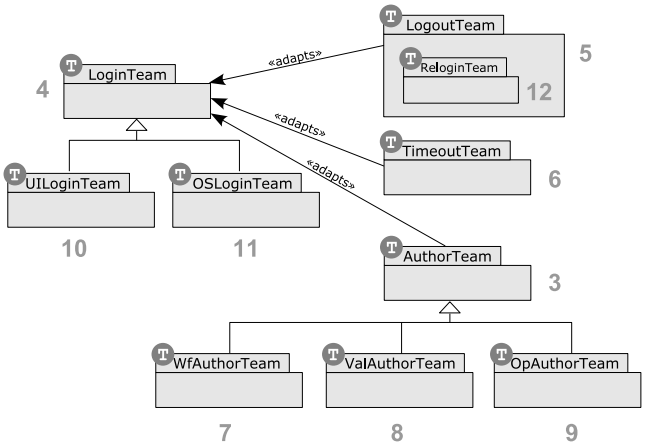


Figure 3: Architecture of the security component

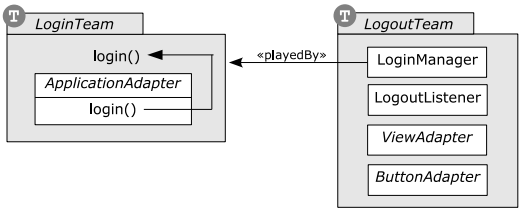


Figure 4: Internal structure of *LoginTeam* and *LogoutTeam*

mapping (b) from Figure 1. For subfeatures of *authentication*, mapping (e) is used. For subfeatures of the features *login* and *authorization*, mapping (c) applies. *ReloginTeam* is a nested team inside *LogoutTeam*, following mapping (d). Mapping (f) is used for all features that crosscut other features, e.g. *LogoutTeam* adapts *LoginTeam*.

Internal Structure of the Security Aspect

Since each of the features of the security component must be bound to different points in the base system, we need to refine the internal structure of the teams, that is, define the roles that are played by elements of the base system.

The internal structure of two of the teams is given in Figure 4: the structure of the team implementing the login feature (*LoginTeam*) and that of the team implementing the logout feature (*LogoutTeam*). Note that finally the *ReloginTeam* was refactored to the more general role *LoginManager*.

Note the naming conventions of the roles: roles that are played by other *teams* of the security component are named *manager* roles (e.g. the role *LoginManager* of team *LogoutTeam* that is played by *LoginTeam*). Roles that adapt the base system are called *adapter* roles (the role *ApplicationAdapter* of the *LoginTeam* and the roles *ViewAdapter* and *ButtonAdapter* of the *LogoutTeam*). All other roles are

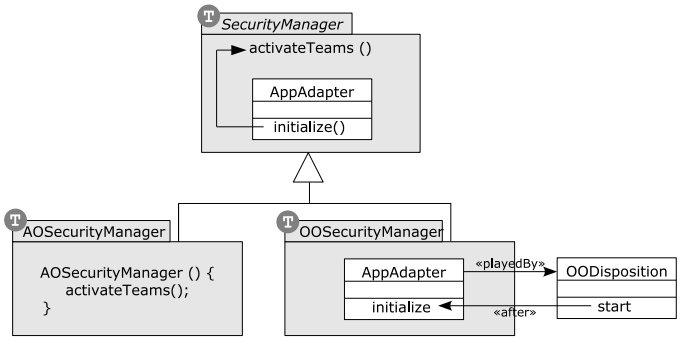


Figure 5: Dependent Activation pattern

played by normal Java classes which implement some functionality of the security component that cannot or need not be implemented in a role or a team, e.g. the role *LogoutListener* in team *LogoutTeam* which is played by a listener class that processes events from logout buttons.

The *LoginTeam* implements a variant of the *Dependent Activation* pattern [21]. The *Dependent Activation* pattern allows to activate teams with different activation contexts (e.g. at application start or after an initialization) without writing different code on the abstract level. For team activation at application startup, the constructor of the concrete team simply calls its activation method. If aspect activation should be performed only after an initialization of the base application, a callin binding must be declared on the concrete level. The structure is illustrated in Figure 5 showcasing the *SecurityManager* team which is responsible for the activation of all teams that are implementing selected features.

Similarly the *LoginTeam*'s functionality can be triggered either at application start or after a set of initialization methods are performed (like initializing the GUI or connecting to a database), i.e. by calling the login method directly from the team's constructor or by binding the role *ApplicationAdapter* in a subteam and connecting its login method to an appropriate base method.

The *LogoutTeam* has four roles. The first of them, the *LoginManager* holds the reference to (adapts) the *LoginTeam*, following the design of the influences-relationship. The next two roles *ViewAdapter* and *ButtonAdapter* must be bound to the base application on the concrete level. The *ViewAdapter* is responsible for re-initialization of the view after logout and re-login. The role *ButtonAdapter* is a role that implements fine-grained features, here the functionality that adds a logout button and a menu item to the GUI of the base system to allow users to logout. The last role *LogoutListener* is only used to notify other roles in the team, e.g. the *ViewAdapter* role, when a logout event is fired. It would have been possible to implement roles with different responsibilities in different nested teams in the team *LogoutTeam* but as there are only four roles we decided against nested teams in this case which also included to replace the nested team *ReloginTeam* by a role.

4 APPLICATION ENGINEERING WITH OBJECT TEAMS

Deriving a concrete product instance from the product line infrastructure should merely be a selection process based on the feature model. However, if the product line is an aspect itself as in the case of the security example, an additional step of binding the instance to a base application is necessary.

Instantiation Process

The general instantiation process consists of selecting the features and hence the corresponding teams, and of specifying instantiation and activation for these teams. For instantiation and activations, there exist several different possibilities:

- All selected teams are instantiated via launch configuration. This kind of feature selection is only applicable if all features are represented by single team instances. If more instances of one team are needed, one of the alternatives below needs to be chosen instead. This is e.g. the case if a cardinality is specified for a feature in the feature model.
- The activation follows the *Dependent Activation* pattern. The activation is controlled by one additional team, the *ActivationTeam*. This team is activated at start-up, and is responsible for instantiating and activating all teams that implement features.
  This solution can control team (feature) activation in a flexible way and also manage feature dependencies.
- In an even more flexible solution, each feature or each set of features can be controlled by its own activation manager. These activation managers are activated via launch configuration or via a global team that activates all activation managers. The latter offers a scalable architecture where each activation manager can activate a set of other activation managers which activate a set of features or again another set of activation managers.

If the product line instantiation is to be bound as an aspect, an additional step has to be performed between selection and instantiation. After features are selected, a concrete level is generated, consisting of skeletons of subteams of all selected teams. These skeletons only include the set of inherited roles with no additional functionality and no binding. These team skeletons are connectors that can be bound to the base system in the next step.

Binding to the Base System

The last step of deriving a product from the product line aspect is to bind the product line aspect to a base system. This step is specific to our approach —

"normal" product lines don't need any special binding to a base system. Note that we use the term "binding" here, because this kind of integration is not a standard object-oriented integration nor an integration using techniques of an object-oriented framework. Instead, it uses non-invasive techniques of aspect-oriented programming languages, and does not require code to be written.

As we have already derived the connector team skeletons, we only have to define the base classes of the roles in the connector teams and some method bindings:

- Callin bindings trigger all unreferenced methods.
- Callout bindings delegate abstract methods of the role to its base.

Both kinds of binding are supported by the Object Teams binding editor which generates the bindings after we have visually connected role and base methods in the editor. For both kinds of bindings, we apply patterns from [21] to be flexible in the binding process.

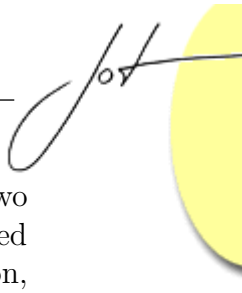
For callin binding, we use the *Feature Selection* pattern[2] [21]. Selecting implemented features on the abstract level is done through callin-binding the methods that implement the features to base methods. Unbound methods have the effect, that the corresponding features are not supported by the derived product.

For callout binding, we use the *Uniform Role Access* pattern [21]. Object Teams allows to hide the implementation of an interface from the client. In contrast to object-oriented languages, the implementation cannot only be provided by a subclass (or here better: a subaspect) of the abstract class, the implementation can also be delegated to the base class of the given aspect (here: role) via callout binding. This abstracts away from how and *where* a given interface is implemented. We use this pattern because sometimes we have to implement a bit of functionality in the connector team. This is mostly the case if we cannot find a suitable binding point in the base system, for example if a requested functionality is not provided by the base system. Currently, the OTDT's binding editor is not "feature-selection-aware" and allows to bind any method of the base system, i.e. also features that are not selected for the given product. For future versions, we plan to extend the binding editor to only provide the binding of selected features (*selective binding editor*).

Instantiation of the Security Product Line

In our case study, security was added to a disposition application for monitoring stock levels and creating orders. This application exists in an object-oriented variant and an aspect-oriented implementation using Object Teams. Both variants have been developed by the industrial partners in the TOPPrax project [23]. Here, we describe the instantiation and binding of a concrete product from the security product line aspect to the aspect-oriented variant.

[2] Note that although this pattern is called *Feature Selection* it is not a general solution for feature selection since only fine-grained subfeatures that are implemented as roles are selected.



We concentrate on the two features *login* and *logout*, which were mapped to two teams in the product line, and for which two connector skeleton teams are derived now: the team *AODispoLoginTeam* and *AODispoLogoutTeam*. For the instantiation, we follow the Dependent Activation pattern and add a team *AOSecurityManager* that activates these two teams. In the connector teams, we describe the bindings but also have to specify initializations required for login and for GUI configuration. The team *AODispoLoginTeam* is responsible for initializing the used login module in the team constructor. The *login* method that calls the login function is called after the GUI is initalized. The abstract team *LogoutTeam* defines two methods in the role *ButtonAdapter* for adding logout buttons and menu items to the GUI. This functionality is selected by applying the *Feature Selection* pattern to the concrete instance of the security component in the team *AODispoLogoutTeam*. This means that the add button methods are executed via callin binding after the relevant part of the GUI is initialized. For the disposition component variant, both a logout button and a menu item are added. The team *AODispoLogoutTeam* also has to specify some initialization code for the configuration of buttons.

We bound the same instantiation also to the object-oriented order system. Here the difference was that we only needed a logout button, but not a menu item. We also instantiated the same features, i.e. logout and login, for the object-oriented variant of the disposition. While the bindings were different in some cases, some of the initialization code was the same as for the aspect-oriented variant. Therefore, we subsequently factored out the commonalities as an additional layer.

5 DISCUSSION

Our approach aimed at improving the alignment of mandatory and variable features with implementation modules in order to support seamlessness in a model-driven fashion. The improved alignment should also facilitate the selection and instantiation process. Furthermore, we proposed patterns for achieving flexible binding possibilities for a product line instance that is to be used as an aspect.

Object Teams for Product Line Infrastructures

We applied the approach in developing a flexible security solution. Either there was a one-to-one correspondence between features and teams or between features and roles contained in teams, or features could be merged with super features in the case of features that do not bear functionality themselves. A one-to-one mapping for all leaf features could be established, which facilitates the instantiation. The security product line consists of 12 teams with a total of 46 roles. The feature model also contained influence-relationships. They were mapped to adapts-relations in the implementation, avoiding tangling and scattering due to influence-relationships.

While we were implementing a security *aspect* with Object Teams, Object Teams

itself is not restricted to that. Teams and their roles do not need to be bound to a base application but can be used like ordinary classes.

Implementing the product line with Object Teams has further advantages over traditional object-oriented approaches. The product line can be built in a *continuous* process, starting from a concrete product. As the Object Teams language supports all stages of genericity that are necessary to implement a product line, transitions between different technologies can be avoided. Generative techniques, e.g. using Frames [15], do not provide a smooth transition starting from existing code.

Most of the described steps from the feature model to a concrete product can be performed using generative techniques. When the whole tool chain is implemented, engineering of a product line aspect and instantiating a concrete product line can be done using model-driven techniques.

Aspects as Product Lines

It was feasible to develop security as a complex aspect that can be bound to base applications that have been implemented not only with object-oriented techniques but also with aspect-oriented techniques. The security component adapts the disposition component's controller which adapts the disposition view and the disposition business object layer. Our case study has shown that Object Teams scales very well in that respect. Using aspect-oriented techniques for binding security to an existing applications avoids that the existing application has to be changed and that security is scattered over a number of classes as security is typically a feature that crosscuts many other functionalities. The same technique of selecting features from the product line applies to the binding of the product line aspect to a given base component.

In most cases, the instantiation process only comprises the definition of a set of connectors using the Object Teams *binding editor* to bind features to a base application. In cases where the signature of a base method is not compatible with that of the role method to be bound, a parameter mapping has to be specified.

Sometimes, additional implementation can be necessary to adapt features of the product line to special needs of the base application. The only cases we encountered are the *ActivationTeam* where a team constructor must be implemented (or can be generated) when using the start-up activation and the *LoginTeam* and *LogoutTeam* where some initializations are done in the team constructor but can also be configured using a configuration file. We also created the resources for the internationalization and implemented the configuration of the buttons in the *DispoLogoutTeam* and some initialization for the login process (here: which login module to use) in the *DispoLoginTeam*. The fact that we had to implement extra code in the connectors does not affect the alignment, though.

The security component is able to adapt different base systems, as we have shown with different other base systems in the context of the case study. Moreover, we agree

with [18] that implementing a security solution using aspect-oriented programming techniques is more flexible than container-managed security.

We observed that there are even commonalities during instantiation and binding that could be further exploited. In our case study, we factored out a common layer for binding to the aspect-oriented and the object-oriented variant of the disposition application.

The paradigm used for creating a concrete instance of a product line is a *declarative* one that is smoothly integrated with the overall imperative paradigm. The enabling of the selected features typically requires to establish callin and callout bindings by writing declarative code. A special graphical binding editor eases this task. It must be noted that these bindings are added on top of the implementation code; they do not tangle the handwritten code. Also, the generation of skeletons and the binding to the base application could be automated.

Evolution of Product Line and Base System

Managing the evolution of a product line asks for how the concrete products are affected by a change of the product line they are derived from. As there exists a tracebility from product line to concrete derived products, changes of the product line can be traced to modular incremental changes in the concrete product. This facilitates the maintance of the product line. For evolution, three cases have to be considered, the addition of a new optional feature, the addition of a new mandatory feature, and the change of existing features. We have developed different changes and enhancements of the security component and the adapted base systems to see how this affects the product line, concrete product line instances and base systems. Examples for changes are the introduction of a new authorization kind (optional feature) or the modification of the login process (mandatory feature).

The addition of an *optional* feature has no effect on the concrete product line instances nor the base systems except for the case where the new feature will be selected and integrated into an existing product line instance. In this case, only a team implementing the new feature needs to be created as well as a connector-team to bind the feature to the base system. Additionally, a suitable activation mechnism needs to be provided. Adding a *mandatory* feature can lead to a lot of changes in existing products derived from the product line depending on the grade of interaction between existing features and the new feature. *Changing* existing features can result in changes in both the product line instances and the base system. As base system and aspect are coupled via interfaces, only a change in an interface potentially leads to changes in the base system but normally, only the binding must be adjusted to the new interface. This also holds for changes in the base.

If the product line is an aspect, one has to consider the problem that the set of selected joinpoints can potentially change during evolution. A solution to the latter problem could be to perform the evolution (e.g. tool-supported refactoring) of both

aspect and base system together. A lot of research is done in this area, for example in [24, 7]. But in the case of a product line, this is not always the right solution. Often, the product line itself is developed and maintained independently from the users of the product line which implement and maintain the concrete product line instances.

6 RELATED WORK

In this section, we compare our work with other approaches aiming at the improvement of software product line engineering.

Sochos et al. [19, 20] also identify the problem that features are often scattered and tangled in system components and suggest a stronger mapping between features and the implementation architecture. Their FArM (Feature-Architecture Mapping) method iteratively transforms an initial feature model to derive architectural components. Several transformations suggest the integration of (crosscutting) features into existing functional features by enhancing their specifications. Even if the transformation decisions are documented in traceability links, this integration causes a tangling of originally separated features. In contrast, the more flexible aspect-oriented mechanisms of our approch allow to continuously preserve the separation of features down to the implementation.

Framed aspects [15] combine aspect-oriented programming with frame technology. Aspect-oriented frames allow to define placeholders for several selectable features. A specification defines the selected features. The binding process is based on a set of composition rules that define the instantiation of the given frames. The placeholders try to overcome specific problems of AspectJ, an aspect-oriented extension to Java. In AspectJ, there is no inheritance between advice in aspects. This means that aspects can only be specialized by redefining pointcuts and abstract methods. The inheritance hierarchy is limited; only abstract aspects can be specialized. It is also not possible to refine different concerns together, since this is not supported for aspects. In our approach, instead of defining a specification, a connector is generated based on the features selected, and composition rules are provided by the mapping rules of features to Object Teams. While in framed aspects a feature is mapped to a frame, in Object Teams a feature is mapped to a team.

The Pure Variants approach [22] aims at product line development for constraint environments. A variant management tool is used for feature selection while the feature implementation is based on AspectC++, an aspect-oriented extension to C++. Like in our approach, the configuration of a product line instance is reduced to the creation of aspect bindings. Compared to ObjectTeams, AspectC++ (like AspectJ) has no concept of collaboration supporting the realization of features (aspects) with more complex structure and behavior. Also the focus of [22] is more on generating small and efficient code than on reusability of the aspects implementing the features.

There is a long tradition of using mixin composition for realizing software prod-

uct lines [5]. Team inheritance in Object Teams is related to mixin layers by its focus on collaborations and collaboration inheritance. However, using mixin composition alone it is not possible to develop truely independent views since the final mixin composition statically creates one large structure which must hold all individual features. There is no mechanism for bridging mismatches between different views. Object Teams add aspect binding as a second dimension of composition with elaborate support for independent views and their composition. By realizing individual features as views (aspects) a far better decoupling can be achieved.

More recently, [14] has evaluated several recent technologies regarding their suitability for feature-oriented development. Their results are mostly positive, which means advanced modularization techniques are indeed very useful for feature-oriented development. In our work on product lines, we identified the concept of roles as playing a central role in modeling and implementation. The explicit and rich support for roles is missing from those approaches discussed in [14].

The work in [2, 3] takes feature-oriented programming a step further by integrating aspects with mixins in order to improve feature-oriented programming. The finding is that the integration unites the crosscutting facility of aspects with the facility of mixins to define individual adaptations. However, in the mixin layer, features are implemented in a layered way with a combination of features on different layers whereas our approach allows to combine feature on the same layer and to refine features independently from each other. No layered architecture is needed.

The relevance of model-driven approaches in product line engineering was already shown in traditional software development (see e.g. [12]) and has now reached aspect-oriented software development, as research projects in the area show. In [1], the importance of generative techniques in product line engineering is stated.

7 SUMMARY AND OUTLOOK

In this paper, we have proposed an approach for using a feature model to drive the development and instantiation of a product line. In order to improve the alignment of crosscutting features with code, we have mapped features to Object Teams. We evaluated the use of Object Teams for developing a product line with a security aspect.

In most cases, mandatory as well as optional features were mapped to individual implementation modules in Object Teams. This provides ideal traceability, supporting maintainability and evolvability. An instance of the security product line is a complex aspect that can be bound to arbitrary base applications using declarative binding mechanisms of Object Teams. The bindings are contained in a layer of so called *Connector* teams. Only a few additional implementations were needed in the connectors to realize specific initializations.

Using object-oriented techniques instead to build a product line requires to create

a new module for each combination of variants, as there is no mechanism with which a variant can be superimposed on a default module. This situation is alleviated by using design patterns. But patterns still require that hook methods are present in the default module.

Instantiating a product line can be more than just selecting features but may also require providing additonal parameters for *values* (e.g. timeout duration, installation paths), *algorithms* (e.g. an encryption algorithm) or *required external components* (e.g. a database system). With Object Teams, we are flexible enough to provide these functions through direct implementation in the connector teams as in the authentication feature or via configuration files as done in the authorization feature.

The security component has shown flexibility using a set of patterns that improve reusability of aspects. These patterns support the activation, implementation and binding of aspects and especially aspects that themselves are product lines.

Currently, the implementation of the instantiated product line contains all variants in the abstract teams. In the future, an optimizing compiler could easily remove all code that is not used for a specific product line instance in order to produce lean systems. The Object Teams development tooling (OTDT) facilitates the derivation of a concrete product and its bindings to the base system at this stage. Future enhancements that add support for product lines can improve the instantiation of a product line aspect.

The proposed approach can be ported to other languages and frameworks that support the ideas of aspectual components [13] supporting role-based and collaboration-based modularisation mechanisms (e.g. CaesarJ [4]).

REFERENCES

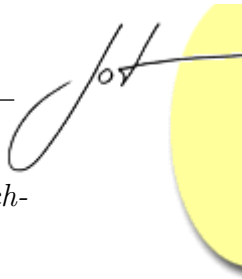
[1] M. Anastasopoulos, T. Forster, and D. Muthig. Optimizing Model-driven Development by Deriving Code Generation Patterns from Product Line Architectures. In *OOSE'05, at Net.ObjectDays'05*, Erfurt, 2005.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In *AMSE'05, at ECOOP'05*, Glasgow, 2005.

[3] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: Aspects and features in concert. In *ICSE'06*, Shanghai, 2006.

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135 – 173. Springer, 2006.

[5] D. Batory and Y. Smaragdakis. Building Product-Lines with Mixin Layers. In *Workshop on Object Technology for Product-line Architectures, at ECOOP'99*, Lisbon, 1999.



[6] K. Czarnecki and Eisenecker U.W. *Generative Programming: Principles, Techniques and Tools*. Addison-Wesley, 2000.

[7] J. Hannemann. Aspect-Oriented Refactoring: Classification and Challenges. In *LATE'06, at AOSD'06*, Bonn, 2006.

[8] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Net.ObjectDays'02*, volume 2591 of *LNCS*, Erfurt, 2002. Springer.

[9] S. Herrmann and C. Hundt. ObjectTeams/Java Language Definition. http://www.objectteams.org/def/0.9/index.html.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353, Budapest, 2001. Springer.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect Oriented Programming. In *ECOOP'97*, number 1241 in LNCS, pages 220–243, Jyväskylä, 1997. Springer.

[12] S. D. Kim, H. G. Min, J. S. Her, and S. H. Chang. DREAM: A Practical Product Line Engineering Using Model Driven Architecture. In *ICITA'05*, Sydney, 2005.

[13] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical report, Northeastern University, Boston, April 1999.

[14] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP'05*, volume 3586 of *LNCS*, pages 169–194, Glasgow, 2005. Springer.

[15] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR'04*, volume 3107 of *LNCS*, pages 127–140, Madrid, 2004.

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

[17] S.Herrmann, C. Hundt, K. Mehner, and J. Wloka. Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In *DAW'05, at AOSD 2005*, Chicago, 2005.

[18] P. Słowikowski and K. Zieliński. Comparison Study of Aspect-Oriented and Container Managed Security. In *AAOS'03, at ECOOP'03*, Darmstadt, 2003.

[19] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Net.ObjectDays'04*, volume 3107 of *LNCS*, pages 138–152, Erfurt, 2004. Springer.

[20] P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In *ECBS'06*, Potsdam, 2006.

[21] D. Sokenou, K. Mehner, S. Herrmann, and H. Sudhof. Patterns for Re-usable Aspects in Object Teams. In *NODe'06, at Net.ObjectDays'06*, Erfurt, 2006.

[22] O. Spinczyk and D. Beuche. Aspect-Oriented Product Line Development in Constrained Environments. In *RICE'03, at OOPSLA'03*, Anaheim, 2003.

[23] TOPPrax Project. http://www.topprax.de.

[24] J. Wloka. Towards Tool-supported Update of Pointcuts in AO Refactoring. In *LATE'06, at AOSD'06*, Bonn, 2006.

ABOUT THE AUTHORS

Christine Hundt is a Ph.D. student at Technische Universität Berlin. Her research interests are aspect-oriented software development, optimization of aspect-oriented systems and virtual machine support for aspect-oriented program execution.

Katharina Mehner works for Siemens Corporate Technology since 2007. Her current research areas include software product line engineering, service-oriented architecture and aspect-oriented software development. From 2004 to 2006 she worked at Technische Universität Berlin in the TOPPrax project. She holds a PhD in Computer Science from University of Paderborn.

Carsten Pfeiffer is a software engineer at GEBIT Solutions in Berlin, Germany. Stimulated by his thesis and the work at his previous employer, the Fraunhofer FIRST research institute, his interests include advanced software modularization and composition concepts.

Dehla Sokenou finished her PhD thesis in 2005 at Technische Universität Berlin, focussing on model-based testing techniques for object-oriented software. Since 2006, she is Senior Software Engineer at GEBIT Solutions. Her research interests include model-driven requirements engineering and testing.

This work has been supported by the German Federal Ministry for Education and Research (BMBF) under the grant 01ISC04 (Project TOPPrax).