Patterns for Re-usable Aspects in Object Teams[*]

Dehla Sokenou
GEBIT Solutions GmbH
Cicerostraße 37
D-10709 Berlin
dehla.sokenou@gebit.de

Katharina Mehner, Stephan Herrmann, Henry Sudhof
Technische Universität Berlin
Softwaretechnik, Sekr. FR 5-6
Franklinstr. 28/29, D-10587 Berlin
{mehner|stephan|hsudhof}@cs.tu-berlin.de

Abstract: Aspect-oriented software development still lacks practical evidence. While aspects are claimed to be useful in adapting existing applications there is also first evidence that they might themselves be re-usable. We present results from two case studies with the aspect-oriented programming language ObjectTeams/Java that investigate the re-usability of aspects in developing a security framework. During the development of the framework we have identified patterns for re-usable aspects that increase the flexibility when applying a framework to a given application.

1 Motivation

Aspect-oriented software development provides new means for modularizing software and thus aims at improving the understandability, re-usability, extensibility, adaptability, and evolution of applications. While purely academic research in this area suggests a major benefit the actual practical evidence is still rare. The joint research project *TOPPrax* [TOP] was initiated to demonstrate by means of industrial case studies that aspect-oriented software development has reached a level of maturity where it is able to master complexity of modern software systems. Based on direct comparative studies of aspect-oriented and object-oriented software development, TOPPrax provides an evaluation basis. In order to make practical deployment more realistic, concepts, tools, and methodology have been completed in parallel to the case studies.

The chosen aspect-oriented approaches Object Teams [HH] and Caesar [MO03] go beyond approaches like AspectJ [KHH+01] by treating aspects as first class citizens including inheritance and polymorphism and by supporting aspects with a rich internal structure through encapsulation and refinement of collaborations.

The main focus of the TOPPrax case studies is on re-usability and extensibility. Here, we present some results from the case studies carried out using Object Teams. The theme of the case studies presented here is security. These case studies investigate not only, whether applications written in Object Teams can be easily extended with a security component, but also whether an aspect-oriented security component can be designed for re-use.

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

The test case for the case studies presented is another TOPPrax case study devoted to the implementation of an object-oriented and an aspect-oriented variant of a disposition component, an ERP (enterprise resource planning) system for monitoring stock items and generating order proposals. Based on this component, a *feasibility study* aimed at investigating the feasibility of implementing re-usable aspects with Object Teams. As one of its effects this case study helped to consolidate the language and its tools. It was also a comparative study that investigated the use of the language JAsCo [SVJ03]. This comparison will not be discussed here. The *security case study*, built on top of the disposition, investigated how a fully functional security component can be designed for re-use.

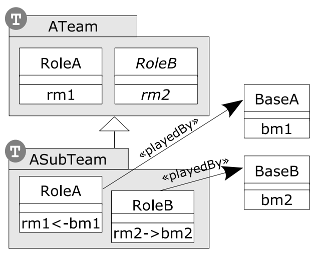
The security component should not only be usable with the two variants of the disposition but also with other ERP systems. Different systems and their given structure might impose different requirements on the security component. Therefore, this component should capture the common core while permitting adaptation for specific requirements. To this end, we have designed and implemented a framework for the security component using aspect-oriented technology. The common core is implemented in a generic way using aspects and can be stepwise refined using aspect-oriented technologies. The aspect-oriented framework does not only use aspects to structure the framework internally, but aspects play a central role in connecting the framework with a given application. Of the numerous patterns identified throughout the development of the framework, here, we focus on those patterns that guide the developer towards re-usability of aspect modules.

The paper is structured as follows. In Section 2, we give a brief overview of the Object Teams programming model. The security case study is introduced in Section 3. Section 4 presents the patterns for re-usable aspects.The case study on re-use feasibility is presented in Section 5. Section 6 discusses the re-use of aspects and synergetics with established re-use technologies. Section 7 compares our approach to related work. In Section 8, we conclude and give an outlook to future work.

2 A Brief Introduction to the Object Teams Programming Model

Object Teams [Her02] is an aspect-oriented programming model that combines several concepts of other programming paradigms and techniques. It introduces a new module concept, the *team*. A team is a package that groups classes (see Figure 1). At the same time, a team has class features, including 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 *roles* that 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 have only 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 can intercept method calls to its base object and execute a role method. Callin bindings can be of type *before*, *after*, or *replace*, meaning that the role method is executed before, after or instead of the base object's method it intercepts. In aspect-oriented program-



```
public team class ATeam{
   public class RoleA{
      public void rm1(){...}
   }
    abstract public class RoleB{
      abstract public void rm2();
   }
}
public team class ASubTeam extends ATeam{
   public class RoleA playedBy BaseA{
      rm1<- after bm1;
   }
   public class RoleB playedBy BaseB{
      rm2 ->bm2;
   }
}
```

Figure 1: Object Teams Example

ming, the elements referred to in callin bindings are called *join points* [KHH^{+}01].

- A role class can specify *callout* method bindings. This allows role instances to forward method calls to base methods. Aside from methods also fields of a base class can be made accessible via callout bindings.

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, not only ordinary classes can be bound by a role but also team classes and role classes.

The adaptation 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 have no effect. 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. If multiple callins affect the same join point the order of execution is determined by the order of team activations in conjunction with precedence declarations within a team if needed. In addition, guard predicates allow the execution of callins to be further restricted.

Object Teams supports bindings to explicitly listed features of a given base class. A new pointcut language for binding a role method to a set of join points selected through advanced query mechanisms is being integrated into the tool-suite for Object Teams.

A more comprehensive introduction to Object Teams is given in [Her02]. An exhaustive description of ObjectTeams/Java, which realizes Object Teams for the host language Java, is given in the official ObjectTeams/Java language definition (see [HH]).

3 The Security Case Study

The security component requires a generic design and implementation that can be refined to fit the needs of the adapted base systems. Before we present the security component we introduce the disposition components, to which the security component is applied.

3.1 Overview of the Disposition Components

The aspect-oriented and the object-oriented disposition components fulfill the same requirements. Each monitors items on stock, supports manual and automatic creation of order proposals, evaluates order proposals, and allows orders to be made. The underlying domain model of both components covers relevant properties of stock items and models various delivery conditions of suppliers.

Both implementations share the user interface and the persistence layer which have been implemented using state-of-the-art object-oriented technology. The user interface has been generated using a GUI modeling framework. The persistence layer of the disposition component has been implemented using a persistence framework. Both frameworks are part of the commercial framework TREND for model-driven development [GEB].

For the aspect-oriented part of the case study, the required functionality has been implemented using ObjectTeams/Java. Teams have been used to encapsulate workflows, to wrap the persistence layer, and to encapsulate a database transaction. The system design follows the model-view-controller architectural style, where an excellent separation between the three parts is achieved using the aspect-oriented capabilities of teams and roles. The object-oriented version has been implemented in pure Java.

Both disposition components have not been designed with a security layer in mind.

3.2 Security Requirements

The security component is intended to provide access control by user authentication and authorization. A simple user management which stores login names and passwords is assumed. For the authorization it is assumed that access restrictions for each user can be specified. Authentication and authorization are the functional requirements detailed in the following. Authorization is dependent on authentication. If only authentication is requested, the authenticated user has complete access to the application.

The minimal requirement for authentication is the *login* feature. How the login is carried out, e.g., through a user interface or through operating system user identification is left open. The login feature does not necessarily imply logout. In the simplest variant, logout is implicit in the shutdown of the application. An optional *logout* feature supports to explicitly log out from the system. After logout a new user or the same must be able to login. Optionally, a *timeout* is available to logout a user automatically after a configurable time span has elapsed without user interaction. Logout and timeout both require the login feature. Logout and timeout do not require each other, however the absence of logout has an implication on the timeout. If no explicit logout is available, the timeout shuts down the implementation, otherwise, a new user can login. Logout and timeout are not supported if the login is realized by operating system user identification.

Authorization is supposed to cover business objects containing data but also workflows. Different access rights and different user roles are distinguished.

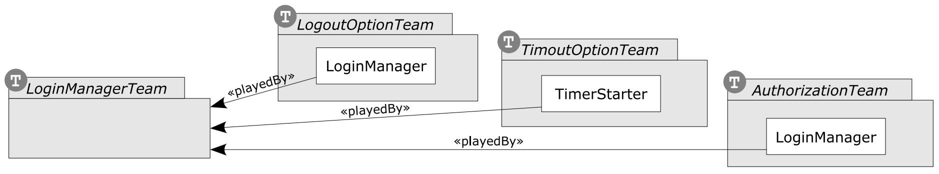


Figure 2: Teams defining contexts for security component

3.3 Security Design Issues

The generic solution for the authentication part of the case study consists of four modules, each encapsulated by using the team concept (see Figure 2). The first team (`Login-ManagerTeam`) is the base for all other teams. It manages the login process. The two optional features are implemented by the teams `LogoutOptionTeam` and `Timeout-OptionTeam`. Authorization is covered by the `AuthorizationTeam`. Using the team concept each feature becomes easily traceable.

The presented teams are abstract and implement basic functionalities of the security component. They can be refined independently from each other to adapt a concrete system. The only references between teams are on the abstract level (abstract coupling). The `LogoutOption-Team` and the `TimeoutOptionTeam` depend on the `Login-ManagerTeam`, e.g. they might need information about the login context. The `Login-ManagerTeam` is independent of the two other teams and should not need to make its internal structure public. Through a `playedBy` relation, the `LogoutOptionTeam` and the `TimeoutOptionTeam` can access features of the `LoginManagerTeam` that have not been made explicitly visible (decapsulation) and they can also adapt the behavior of the `LoginManagerTeam` when needed. The `LoginManagerTeam` can thus be designed unaware of the two other teams. Similarly, the `AuthorizationTeam` is dependent on the `LoginManagerTeam` but not vice versa.

The teams for the options logout and timeout are independent. If one of them has no concrete subteam or the concrete subteam is not activated, the given option is not available. The team concept even allows to activate and deactivate the options at runtime; for example, a user-dependent timeout is easy to realize by deactivating or activating the timeout option for a specific user.

The requirements lead to a framework-like design of the abstract solution of our security component. The abstract implementation should be flexible enough to be adapted to different base systems. The strategies for implementing the concrete level and adapting the base component vary significantly between systems (see also next section). In the end, we have implemented a three step refinement of teams for the security solution applied to the disposition components: the generic solution, the disposition-specific solution and a solution depending on the variant (object-oriented or aspect-oriented). In the process of refining the abstract security solution, we have detected a set of patterns that allow a flexible refinement on the concrete team level. The next section introduces the main patterns we have found in our security case study.

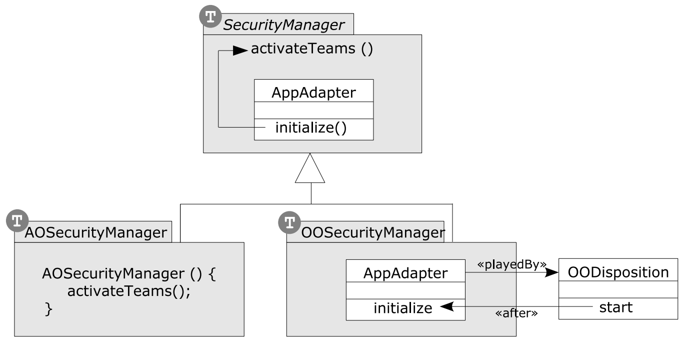


Figure 3: Dependent Activation

4 Patterns for Re-usable Aspects

This section presents some patterns we have identified while implementing the security case study. Another pattern identified while implementing the feasibility study is introduced in Section 5. Some of these patterns were also found in other systems that are implemented in ObjectTeams/Java. All presented patterns consist of a generic solution that implements an abstract team, and a specific solution which inherits and refines the generic solution. In the following, we call the generic solution the abstract level, and the inherited and refined solution the concrete level. The patterns are presented by giving a short description of the problem and our solution, followed by an example from the case study and an overview of the general structure.

4.1 Dependent Activation

Different team modules differ in what context they require to be setup, before they can be initialized and activated. While some can be activated right at program start, other teams require that certain initializations happen beforehand. A solution is sought that flexibly supports various dependencies of this kind.

In the security case study, the described problem occurred when implementing the security manager[1]. The security manager is a team that activates all security-relevant teams to offer security for the disposition component. Being developed by two different software engineers, each variant of the disposition has a different start mechanism. We wanted to apply the same generic solution on the abstract level for both variants which lets the developer decide when to activate the teams managed by the security manager.

The solution for this problem is shown in Fig. 3. On the abstract level, the method for initialization –here: `activateTeams`– is introduced in the team.Additionally, a role is implemented –here: `AppAdapter`– with a method that calls the initialization method.

[1]Note: Our security manager which is a team must not be confused with the Java security manager.

At the concrete level, we have essentially two options for refining the superteam. The first implementation is to call the initialization method directly, in the example shown in team `AOSecurityManager`. Typically, the initialization method will be invoked by the team's constructor, like in the example. The team's constructor will in turn be hooked into the application's startup mechanism[2]. In this case, the role `AppAdapter` is left unbound.

The second way of implementation –in the example realized in the team `OOSecurity-Manager`– is to bind the contained role to a class in the base system and let the initialization method be called indirectly via a callin binding to the role method. In the example, the role `AppAdapter` is bound to the base class `OODisposition`, and the method `activateTeams` is indirectly called by intercepting the base method `start` with the method `initialize`. An instance of the `OOSecurityManager` is constructed and activated by the application's startup mechanism.

This solution realizes the required flexibility. When a given state in the base system is reached the initialization method can be triggered by binding it to a method in the base system that represents that state change. A restart or reset can also be implemented using that pattern if the method triggering the aspect is called more than once. A guard can assure that the method will only be called once when resetting is not desired. Both possibilities of implementing the subteams can be combined, e.g. if the team should be initialized on start-up of the base system and should be reset by a given trigger.

The role should not be abstract in the superteam because we do not want to force the subteam to refine this role. To avoid access to the role from outside the team, the role should be declared as protected which for roles defines a stronger restriction than the normal semantics of `protected` in Java. Thus, if not used as a trigger for initialization the role remains unbound in subteams and it is guaranteed not to be initialized from outside.

4.2 Feature Selection

The second pattern resembles the situation of designing a library: A module implements a rich set of functionality whereas specific applications will use only part of it. However, in the case of a re-usable aspect there is no main program that selects features by explicit method calls.

In our security case study, we found such a situation when adapting the disposition GUI to support an explicit logout operation. Different elements like a menu item and/or a logout button need to be added to the GUI. Both possibilities can be implemented on an abstract level but the implementation of the concrete logout team should decide in which way the user can logout.

[2]ObjectTeams/Java supports the addition of aspects to an existing application by simply specifying the given teams in the application's launch configuration.

This is easily solved in Object Teams. The abstract team (see Fig. 4) implements the full set of functionality, here: adding the button and the menu item. The subteam – here: `LogoutOptionTeam'` – is free to bind whichever subset of methods should be used. In the example, callin bindings will cause only the button to be added to the GUI, not the menu item. The method `addMenuItem` is unbound. Thus, it will never be called.

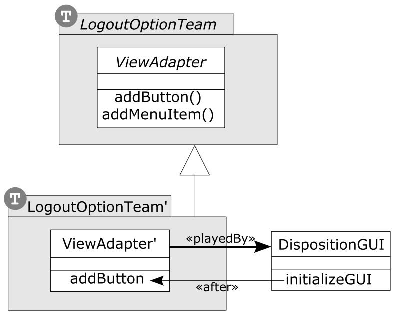


Fig. 4: Feature Selection

There are some advantages of partially bound role behavior. All behavior that can be implemented on the abstract level is realized there. The user of the abstract classes which implements the concrete adaption of a base system can select functionality by simply binding methods or leaving them unbound. If the whole functionality will be used, role methods can be bound by the same base method (in the example, both `add` methods can be bound by the method `initializeGUI`). It is not necessary to find different triggers.

4.3 Uniform Role Access

It is a good object-oriented style to implement against an abstract interface rather than referring explicitly to its implementation. Using Object Teams the realization behind an abstract interface can be provided in two different ways: by implementing a functionality directly or by delegating the call to the adapted base object using a callout binding. Based on the option of implementation vs. delegation an abstract role method can be used to abstract not only from *how* certain behavior is implemented but also from *where* it is implemented. Behind this abstraction a (refining) role is completely free to choose or even combine both techniques for providing behavior. Both kinds of behavior can be accessed in a uniform way.

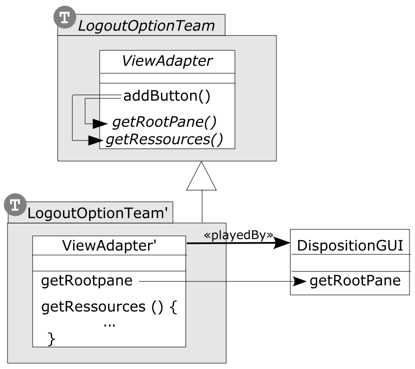


Fig. 5: Uniform Role Access

Fig. 5 illustrates this situation and shows where we use the uniform access to role behavior in the context of the security case study. The `LogoutOptionTeam` is responsible for the logout option. This team has a role `ViewAdapter` that adds buttons to the given disposition view. To implement the addition of buttons, a set of helper methods is declared abstract in the superrole and must be redefined in subroles. The example shows two of these methods. Method `getRootPane` returns the root pane of the disposition view and method `getRessources` returns the ressource bundle to use. Both methods are defined in the same role and can be called on a role instance but the behavior differs. The first method `getRootPane` is delegated to the adapted view object which returns the rootpane of this view object. The second method is implemented completely in the role, no information from the base object is needed.

The uniform view of a role decouples the caller of any role method from the design decision of how an implementation is provided. The developer of the concrete level can decide whether the interface is implemented in the role itself or delegated to the base object. Both ways of implementing the interface can be mixed. Using parameter mappings, the interface of the base method can be mapped to the required interface of the role method enabling the Virtual Restructuring pattern, which is however beyond the scope of this paper. The Uniform Role Access pattern would not be possible in a language that strictly distinguishes between collaboration modules and connectors (see discussion in Sect. 6).

4.4 Reminder Roles

As we implemented the abstract level, we had to decide how to cope with roles that could possibly be needed in the context of a given team but have no functionality nor references on the abstract level. If we do not give a hint in the abstract team, maybe the developer of the concrete level will forget or be confused about the role and believes such a role is not important. Otherwise, we can implement such an empty role but in this case we have to explain that this role has no behavior and no dependencies.

We propose the second alternative. We call these roles *reminder roles*. Their purpose is methodical. They remind the developer that there are roles to implement and bind on the concrete level –we are free to give detailed documentation that explains how to use these roles– and we have a consistent naming of these roles in all subteams, thus, teams are better readable and comprehensible. Reminder roles are normal, but empty roles. If we want to force implementation in subteams, we declare them as abstract but also concrete reminder roles are possible, not demanding for an implementation in subteams.

We found such a role in the team `TimeoutOptionTeam` where the reminder role `View-Adapter` indicates that the actual view should be reset or managed after timeout.

5 The Re-use Feasibility Study

Independent of the main security case study, a comparative case study examining the feasibility of aspect re-use was performed [Sud06]. This study shares several basic considerations with the main case study, such as the use of the TOPPrax disposition system as part of the study's set of base applications and the focus on security aspects, but differs from this paper's main study in its scope and goal. Being a feasibility study the focus was more on exploration rather than on delivering a product. By the feasibility study we assessed

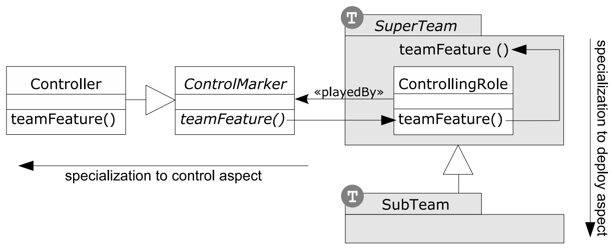


Figure 6: A role acting as controlling role.

the conceptual and technical maturity, especially regarding aspect re-use in oblivious base applications. Several improvements to the language and its tools have been stimulated by this case study, from which the main case study could then benefit.

In the course of the study a set of abstract aspects was designed and implemented. The selected domain included a session aspect to provide a way to propagate user context information in base applications not offering such a functionality and to allow uniform access in base applications which do. This aspect was complemented by aspects for basic authentication and authorization. While were designed to cover different parts of the domain, the target was to do so without causing the aspects to depend on each other.

5.1 Controlling Roles

During the design of the feasibility study's set of aspects, the need arose to allow aspects to be specialized along two different directions. The first direction being the coupling between cooperating aspects and the second the adaption to a concrete base application. To permit the – initially uncoupled – aspects to cooperate, without reducing their ability to be deployed in a concrete environment, a design based on a pattern we call *controlling roles* was introduced. *Controlling Role* stands for a role class bound to an abstract controller class, which declares methods echoing the team's basic features. The controlling role contains method bindings to the controller classes' methods, delegating the calls to the corresponding team-methods. Figure 6 shows the basic layout of this architecture. Note that there are no direct links between the participating entities, removing the need to exchange references during the program start-up. This also means, that all calls of inherited methods on instances of subclasses of the `Control Marker` will be intercepted by *all* active instances of the team. This may require some caution with non-singleton teams. While using a bound role in a high-level aspect component might initially appear counter-intuitive, it actually is a means to modularize dependencies. Essentially the pattern uses the control marker class as a public interface of the aspect. As that class is not directly part of the team, it can be specialized orthogonally to the specialization of the team, even allowing the controlling class to be extended into other team classes. Thus it is possible to use concrete subclasses of the marker to wire the aspects to each other, while leaving the team inheritance to deploy aspects in the concrete base application.

The Controlling Role Pattern was central to the design and implementation of the feasibility study's set of security aspects, which was successfully deployed in several base applications. While the aspects were not explicitly coupled initially, the design allowed the implementation of the inter-aspect communication in re-usable modules. The resulting system was too limited to be considered a framework, but showed that aspects can cooperate to cover a domain, without resulting in an inherent tight coupling between aspects.

The findings of the feasibility study back the main study's notion of striding towards frameworks of re-usable aspects. Similarities to patterns presented in section 4 were identified in the independent designs, indicating the presence of an intuitive set of idioms and patterns for ObjectTeams. Also, we found modularization of connecting code to be possible, enhancing the re-use of such connections.

6 Discussion

As we have seen in our case studies, patterns can be found in aspect-oriented programs. When using aspect-oriented techniques, we go beyond the object-oriented concept of inheritance where subclasses can only override existing methods and add new ones. Object Teams provides even more flexibility by combining inheritance with role concept, callin binding mechanism and callout feature.

An important feature used by most of the presented patterns is the connector concept. A predecessor of the Object Teams model, *Aspectual Components* [LLM99] introduced the distinction into *collaborations* and *connectors*. We have two reasons for not enforcing this distinction by explicit language features: the one concept "team" suffices for both purposes; in terms of conceptual economy, our solution provides the same benefit at a lower price. Secondly, the unification of collaborations and connectors emphasizes flexibility over strictness by allowing the definition of teams which contribute to a collaboration in terms of added implementation and at the same time define connections in terms of role base bindings. We have seen that this flexibility is an essential enabling feature for the design of re-usable aspects.

Thinking a step further, re-usable aspects have the potential to advance established software re-use techniques. The classical issue in software re-use is the separation of commonalities and variabilities among several applications. Decades of object-oriented design provide plenty of experience on how to develop generic implementations prepared for refinement. Still, the development and application of modules that are re-usable in a great variety of usage-contexts is one of the most difficult tasks in software development even today. Two groups of questions need to be addressed to understand these difficulties:

1. What is the nature of variation points? Are they identifiable parts of the implementation? Are they (necessarily?) marked as variation points or can implicit variation points be used? What are the mechanisms to bind a variation to a variation point?

2. What is the overall process of software development? Which roles are involved, how do they communicate? Who defines the rules?

The simplest form of re-use is a library where variation points are limited to explicit method parameters. For some time *frameworks* were considered the preferred technology for re-use in all other settings where libraries are just too limited. The central mechanism in object-oriented framework design is the use of dynamic binding to support methods as variation points. I.e., during framework instantiation application-specific methods can be provided that will be called by code within the framework. While this concept is very powerful and successful in certain areas, frameworks seem to have a problem with scalability.[3]

We believe that the questions from group (2) give the key to the limitations of both libraries and frameworks. Libraries provide fixed implementations of well defined functionality, giving full power to the application developer when to invoke which function and with which arguments. Frameworks, to the contrary, make no final statement on how a given feature is implemented but are quite strict with respect to the overall structure and perhaps control flows within an application.

This results in tremendous difficulties if one tries to integrate more than one framework into the same application [MBF99]. The preferred way of using a framework is to first choose the framework and then develop an application design according to the rules defined by the framework. The dilemma is in fact similar to what has been called the "tyranny of the dominant decomposition" [TOHS99]: no matter where you start, the initial design decision will significantly limit later choices. Much in the vein of [TOHS99] we suggest to consider *composition* as a primary focus in software development in order to allow different structural breakdowns to co-exist within the same application and leaving the integration to a separate software unit.

Coming back to the first group of questions above this means that composition should in fact be considered a variation point. As an example consider our first pattern, Dependent Activation. Usually a framework provides hooks which may be used by applications in order to be triggered during system start for performing application specific initialization. In such cases the application may decide, *what* initializations to perform, but it cannot freely choose the point in time *when* to perform initialization. Using the pattern, the initialization of the application and of the aspect are implemented in complete ignorance of each other. Integration is the sole responsibility of the connector which is implemented as a refinement of the aspect. Note that our proposed solution is even more flexible than the use of abstract pointcuts as it is suggested to be good AOP design. Using our pattern it is not even predefined *whether* any joinpoint within the application will be used as a trigger.

The Feature Selection pattern can be interpreted as a very convenient mechanism to provide what has been called the *framework internal increment*, i.e., speculative functionality as part of the framework that *could* be useful for applications but does not belong to the core structure of a framework. In conventional object-oriented frameworks selecting features from the framework internal increment relies on object instantiation and method calls only. To this our patterns adds new options for selective integration.

[3]One of these problems relates to the fact, that standard object-oriented languages provide means to override methods, but not to override classes. Thus, defining the selection of classes as a variation point is quite cumbersome in those languages. Object Teams overcomes this problem by using *virtual classes* as further specified in [Ern01].

Finally, the Controlling Role pattern spans the team inheritance and the controlling role inheritance dimensions for adding application specific increments.

Using Object Teams with patterns like those given in this paper can break the tyranny of any dominant design decision. Key features to a ”more democratic” software model are:

1. Explicit binding of classes with bi-directional method bindings (callin and callout).
2. Role objects allowing to adapt existing objects without changing their structure.
3. The unlimited option to refine existing classes, no matter if they are base, team or role classes. In AspectJ, e.g., a concrete aspect cannot be specialized further which limits the applicability of our patterns.
4. The ability to compose aspects from aspects by allowing team classes to appear in the position of roles and of bound base classes alike. Unlike in Object Teams, the “advice” construct in many AOP languages can neither be redefined nor used as a joinpoint nor bound to more than one pointcut.
5. Based on the given language features each of the patterns presented defines a new variation point which did not exist as an well-defined variation point before.

7 Related Work

In this section, we compare our approach to related work focusing on aspect-oriented security and re-usable aspects.

Security is mentioned as one of the main applications for aspect-oriented programming techniques since security code crosscuts the application in the same way as other typical aspects like logging. Therefore, a lot of work is done in the field of security and aspects (see for example [Lad03, Bod04, LvdLT04, HWL04, SZ03]). Similar to our implementation, most of the presented work is based on JAAS. Some differences are rooted in the fact that most prior research uses AspectJ as aspect-oriented programming language. Most publications on aspects for security are limited to the purely behavioral part, lacking an integration into the applications GUI. This issue is explicitly addressed by our work.

In [Lad03], both issues of JAAS, authentication and authorization, are addressed by using aspect-oriented programming. For the authorization part, so-called `worker objects` are used to delegate the `proceed` of an `around` advice to an object of type `PrivilegedAction` like it is requested in JAAS. A similar technique is used in the authorization part of our security case study, using a new Object Teams language capability that allows to implement worker objects in a similar, but type-safe, way.

A comparison between container-managed and aspect-oriented security (based on JAAS, too) is found in [SZ03]. The conclusion is that aspect-oriented security is more flexible since container-managed security is limited to features of the container. Both share the advantage of obliviousness [FF00] and can be combined easily. In TOPPrax project, a comparison of the developed security framework with security managed by the TREND framework of GEBIT [GEB] will follow after finishing the case study’s authorization part.

In [HWL04], a security framework is presented. The authors mention that generic security aspects are hard to develop because the adapted applications have different requirements regarding security. We have made the contrary experience. Our approach shows that generic security aspects can be implemented. Different options are implemented in different generalized teams and can be specialized independently for the adapted system.

An approach of using view connectors for implementing a security framework is found in [VPWJ04]. The presented framework is realized using JAC [PSD$^+$04]. View connectors have many similarities with roles in Object Teams. In the presented solution, only wrappers for adapted objects –meaning the definitions of join points– are generic. The approach seems to be less flexible than our approach meaning that features cannot be flexibly combined like in our security implementation. In [VPW$^+$05], this approach has been ported to CaesarJ [MO03]. In addition, the approach is made independent from a concrete authorization engine by using wrappers which only require that the engine used conforms to a basic model of authorization. However, it does not change the overall approach that no framework guides or controls the integration of the domain and the authorization engine.

In [HHUK04], a case study on an aspect-oriented framework for graph traversal is presented. As we have found in our work, the authors mention that new composition techniques provided by aspect-oriented programming help to implement more flexible frameworks. Aspect-oriented frameworks can reduce the complexity of both, the framework and its specialization. Join points are seen as additional hooks for customization. Since this work is focused on join points, it cannot yet be compared with our work directly, as complex join points will be only introduced in the authorization part of our case study.

A collection of re-usable AspectJ aspects is presented in [Isb]. The work discusses deployment and implementation of such aspects. The presented aspects perform isolated functionalities and are thus unlike our notion of aspect frameworks, as they do not cooperate to cover a domain. However, the use of XML aspect deployment descriptors shows similarities to the concept of connectors.

8 Conclusion

Developing re-usable aspects is a challenging task because it breaks the linearity of the software development process. In this approach both the application and the aspect are to be developed independently. Neither the application developer nor the aspect developer are able to make use of knowledge about the other component. In fact the application may not even be prepared for any security aspect. Even more so, the aspect should be prepared for integration and adaptation with a great variety of very different applications. This calls for a kind of generality that has not been witnessed before.

From this requirement the need for new kinds of variation points arises. As we have shown in our paper, the language ObjectTeams/Java is capable of providing new manifestations of variation points within a program. By these capabilities the object-oriented tradition of frameworks is revived and new forms of re-use become manageable. On the other hand, due to the additional flexibility introduced by this language, it is even more important to provide guidance on how to use the new powers with style.

The desired guidance shall be given by collections of idioms and design patterns. We have presented some patterns which have been found in two case studies. We focussed on those patterns that provide the flexibility to make our aspects re-usable. This work is based on earlier experience of aspect-oriented development of the application core where also a number of patterns has been identified. It is important to note that the new patterns make hardly any assumption regarding the implementation techniques used in the application core. In fact, in one variant of our system the core was already implemented using aspect-oriented features. We are not aware of prior work demonstrating to this extent how aspect-oriented programming can be used consistently for a whole system.

The flexibility gained by the patterns in this paper can be grouped in different ways. The Dependent Activation and Feature Selection patterns both relate to receiving trigger events from the application. The Uniform Role pattern and the Controlling Role pattern provide additional flexibility for aspect refinement. The patterns make extensive use of all key features of ObjectTeams/Java, demonstrating the suitability of this language for the design and implementation of re-usable aspects.

We are currently finishing the authorization part of the case study, where we have found even more occurrences of the existing patterns and perhaps will find a few more patterns, too. Although we found the presented patterns in two security-related case studies, we can describe them in an abstract manner. Thus, we are confident that these patterns can be applied to other domains with similar problems. In order to support this assumption, we will apply detected patterns while implementing case studies in other application domains. All these patterns are being collected in a comprehensive pattern catalog.

References

[Bod04] R. Bodkin. Enterprise Security Aspects. In *Workshop on AOSD Technology for Application-Level Security, AOSD'04*, 2004.

[Ern01] E. Ernst. Family Polymorphism. In *Proc. of ECOOP'01*, volume 2072 of *LNCS*. Springer Verlag, 2001.

[FF00] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, ACM SIGPLAN Notices, 2000.

[GEB] GEBIT TREND Framework for Java. http://www.gebit.de/Loesungen/trend_web/Loesungen_trend_en.htm.

[Her02] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays)*, volume 2591 of *LNCS*. Springer Verlag, 2002.

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

[HHUK04] S. Hanenberg, R. Hirschfeld, R. Unland, and K. Kawamura. Applying Aspect-Oriented Composition to Framework Development: A Case Study. In *1st Int. Workshop on Foundations of Unanticipated Software Evolution, ETAPS'04*, 2004.

[HWL04] M. Huang, C. Wang, and L.Zhang. Toward a Reusable and Generic Security Aspect Library. In *Workshop on AOSD Technology for Application-Level Security, AOSD'04*, 2004.

[Isb] W. Isberg. Check Out Library Aspects with AspectJ 5. AOP@Work (14), http://www-128.ibm.com/developerworks/java/library/j-aopwork14.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc of ECOOP'2001*, volume 2072 of *LNCS*. Springer Verlag, 2001.

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. In *Technical Report*, Northeastern University, April 1999.

[LvdLT04] R. C. Laney, J. v. d. Linden, and P. Thomas. Evolution of Aspects for Legacy System Security Concerns. In *Workshop on AOSD Technology for Application-Level Security, AOSD'04*, 2004.

[MBF99] M. Mattson, J. Bosch, and M. Fayad. Framework Integration Problems, Causes, Solutions. *Communications of the ACM*, 42(10):81–87, 1999.

[MO03] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proc. of AOSD'03*. ACM, 2003.

[PSD$^+$04] R. Pawlak, L. Seinturier, L. Duchien, L. Martelli, F. Legond-Aubrey, and G. Florin. JAC: A Framework for Separation of Concerns and Distribution. In *Aspect-Oriented Software Development*, chapter 16, pages 343–369. Addison-Wesley, 2004.

[Sud06] H. Sudhof. Vergleichende Fallstudie über Techniken für wiederverwendbare Aspekte (*german*). Diploma thesis, Technische Universität Berlin, Germany, 2006.

[SVJ03] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proc. of AOSD'03*. ACM, 2003.

[SZ03] P. Slowikowski and K. Zielinski. Comparison Study of Aspect-Oriented and Container Managed Security. In *Workshop on Analysis of Aspect-Oriented Software, ECOOP'03*, 2003.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. of the 21st ICSE*, 1999.

[TOP] TOPPrax Homepage. http://www.topprax.de.

[VPW$^+$05] T. Verhanneman, F. Piessens, B. De Win, E. Truyen, and W. Joosen. Implementing a Modular Access Control Service to Support Application-specific Policies in CaesarJ. In *1st Workshop on Aspect-Oriented Middleware Development*. ACM, 2005.

[VPWJ04] T. Verhanneman, F. Piessens, B. De Win, and W. Joosen. View Connectors for the integration of Domain Specific Access Control. In *Workshop on AOSD Technology for Application-Level Security, AOSD'04*, 2004.