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, October 2007

Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity

Stephan Herrmann, Marco Mosconi, Technische Universität Berlin, Germany

In central fields of software engineering, there are two competing directions of research. Component frameworks and advanced programming languages both seek to improve fundamental software quality properties, most notably: modularity. While both directions have produced specific benefits, harvesting these benefits still requires a trade-off in selecting one of the two technologies. In this paper, we present the integration of the aspect-oriented programming language ObjectTeams/Java into the OSGi component framework. By combining these technologies, the design space for modular architectures of components and applications is enriched with fundamentally new options. As a result, the best of both worlds is available for re-using software modules at realistic scales and for evolving systems over significant periods of time.

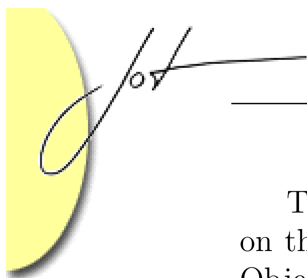
1 INTRODUCTION

In an earlier paper [6], we jointly put forward a vision statement regarding the future of modularity. We claimed that the original "software crisis" was migrating towards a more specific "modularity crisis". Even today, problems of modularity are more urgent than they have ever been due to the growing complexity of software, the increasing demand for re-use and the need for long-term software evolution. Each of these problems, if it occured in isolation, could probably be solved with today's technology. The problem is, to solve all these conflicting demands at the same time.

We have also previously stated [6] that advanced programming languages and component frameworks both overlap in their aim to improve support for modularity, re-use and evolution. Given the severity of this issue, we saw the synergetic integration of the two directions of research as the only way to dispel the approaching modularity crisis.

In this paper, we report on how this vision of synergy from six years ago is now becoming reality. We developed the programming language ObjectTeams/Java [18] which combines aspect-oriented programming with concepts like collaborations and roles. This paper describes the integration of ObjectTeams/Java with the OSGi component framework [19], more specifically Eclipse's implementation of the OSGi standard, called Equinox. This new technology, called OT/Equinox, basically introduces a new kind of inter-component relationship: aspect bindings between Eclipse plug-ins. OT/Equinox has already been applied for the development of the Eclipse-based tooling for Object Teams. This real-world application will serve as a case study in this paper.

Cite this article as follows: Stephan Herrmann, Marco Mosconi: *Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity*, in Journal of Object Technology, vol. 6, no. 9, Special Issue: TOOLS EUROPE 2007, October 2007, pages 105–125,
http://www.jot.fm/issues/issues_2007_October/paper6



The paper is organized as follows: The remainder of this section elaborates on the problem in general. Sections 2 and 3 introduce the essentials of OSGi and ObjectTeams/Java, respectively. The combination of the two is motivated by a real-world example in Section 4 and explained in Section 5. The improvements obtained are illustrated by the example in Section 6 and summarized in Section 7. Section 8 discusses related work. The conclusion in Section 9 summarizes the benefits of OT/Equinox and highlights future work.

Modularity crisis

After decades of successfully building complex systems using object technology, why do we speak of a modularity crisis? What are the obstacles to optimal software structures? We consider three available technologies:

1. Fundamental object-oriented concepts as supported by mainstream programming languages, as well as by growing suites of tools, libraries and frameworks.
2. Component technology defining (a) what a module is and how it is specified, (b) how components are composed to form a system and (c) how fundamental services that are not application-specific are linked into the system [22, 2, 24].
3. Advanced programming languages that extend object-oriented programming towards roles [10], aspects [4] and multi-dimensional separation of concerns [23].

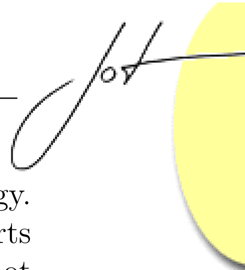
Each of these technologies provides valuable design options. For the purpose of this discussion, we group their achievements into three fields:

Scalability: In software design, structural scalability[1] is achieved by the option of composing a system from modules, which themselves may be composed from smaller modules, etc. Such decomposition draws its strength from clear contracts (interfaces and guarantees) between the components.

Adaptability: Adaptable software modules are open for a wide range of future adaptations, which where not fully anticipated during the module's initial design. As a result, an adaptable module can be re-used in a wide range of contexts *and* it can evolve over time. It is important to be able to exploit both of these dimensions of flexibility together.

Added Value: Software modules may make use of certain services that are orthogonal to the module's logic proper. This should help to separate the application logic from technical concerns such as configuration, life cycle or persistence, which are provided by some part of a module's runtime environment, like, for example, a component container.

[1] The term structural scalability should delineate the discussion at hand from performance issues of scalability.



The bad news is: each achievement is supported only by specific technology. A technology may perhaps help in more than one direction, but none supports them all in an optimal way. On the other hand, the available technologies are not compatible per se: architects are faced with trade-offs between mutually exclusive technologies, a combination of technologies being required to obtain all the desired benefits. For example, scalability is not well supported using classes as the only concept for modules. Thus, an architect may want to group classes into components. Once the step from classes to components has been made, some flexibility is lost: most component models do not support inheritance of components as it is supported by classes. The advanced mechanisms from aspect-oriented programming (AOP) or related approaches, even if integrated with standard component frameworks, also offer flexibility only at the underlying class level, as discussed in Section 8.

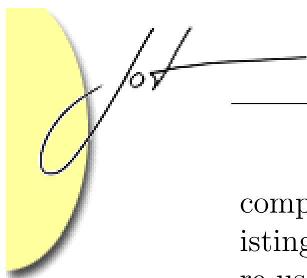
This conflict between scalability and adaptability is not an accidental problem of any particular technology. These technologies are just not designed for free combination. Of course, the largest common denominator is usually the underlying object technology. In this way, components, aspects, roles, etc. could all be mapped into object-oriented structures, thus achieving a minimal compatibility. However, in real life such mapping counteracts every benefit from advanced abstractions and therefore should not be directly used by developers. Instead, we need an integration of technologies *and concepts* such that the same (or similar) concepts can be used across different architectural levels.

The open-closed principle for (re-)usable modules

Conflicts like the tension between scalability and adaptability are not completely new and initial solutions exist. In [17], Betrand Meyer postulates the "open-closed principle", which states that any module should be closed so that it can be faithfully used by clients, while at the same time being open for adaptation in various re-use scenarios. In object-orientation, this principle is realized by the duality of uses-relationships and inheritance. Despite its partial success, it appears that the object-oriented solution falls short in two ways:

1. If classes are the only means for modularization, structural scalability cannot always be achieved as desired.
2. Simultaneously adapting a given module along multiple dimensions is not well supported.

In this paper we present OT/Equinox, which combines component-based, aspect-oriented and collaboration-based development into a uniform programming model. By this combination we lift the open-closed principle from the level of classes to components. The OSGi standard defines how "bundles" are defined as black-box components and how applications can be composed from bundles by wiring these components using export and import declarations. OSGi bundles are therefore *closed*



components. By using the aspect-oriented concepts of ObjectTeams/Java, any existing component can also be *opened* for unanticipated adaptations, thus allowing re-use in situations that were not previously amenable to it.

As the price to be paid for the added flexibility, aspects create new dependencies that could interfere with existing interface contracts. It is therefore a focus of our integration that aspect dependencies do not undermine the given module structure and its guarantees but that aspect bindings are made explicit so as to become an integral part of the component model. Components should thus become aspect-aware, and aspects should become scalable modules.

Most importantly, it is no longer necessary to choose the benefits of either an advanced programming language or a component framework. OT/Equinox enables developers to exploit the best of both worlds in a seamless manner.

2 THE OSGI COMPONENT MODEL

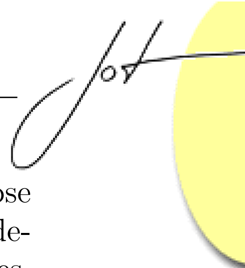
The OSGi standard [19] defines a light-weight component model, which addresses both static and dynamic issues of component composition. In the OSGi terminology, components are called "bundles". The work presented in this paper uses the Eclipse implementation of the OSGi standard: Equinox. Since Eclipse adds some of its own concepts to OSGi, we will at times use the Eclipse term for component: "plug-in".

In OSGi, dynamic support focuses on so-called life-cycle support, which allows to dynamically install, start, stop, update and uninstall a given bundle at run-time without shutting down the application. Bundles are also the primary unit of deployment, which includes the static part of packaging artifacts into a bundle and the dynamic part of retrieving resources from bundles, which completely hides any details of the physical deployment on discs, file systems and the idiosyncrasies of operating systems.

On the static side, a bundle manifest declares the identity of a component and its exports (`Export-Package`) and imports (`Require-Bundle`). Exports are a list of Java packages, all classes contained in these packages being visible to other bundles. Imports refer to whole bundles. Based on these declarations, OSGi guarantees that no bundle can ever access a class from another bundle that is not explicitly made available by suitable export and import. This guarantee is enforced by using appropriately configured class loaders for each bundle.

Equinox adds two levels of visibility: by declaring a package `x-internal`, it is exported without any contractual promises. Classes within internal packages may change or even disappear between versions without notice. A package can also declare friendly plug-ins using the `x-friends` directive, which exports the given package to dedicated plug-ins only.

Exports and imports define a relation between bundles that supports only uni-directional communication. For more sophisticated patterns of communication OSGi



provides the concept of services and a registry for services. For similar goals Eclipse introduced extension points and extensions.[2] *Extension points* can be briefly described as a meta-mechanism, by which each plug-in may declare specific registries, where other plug-ins may announce that they want to contribute to a particular behavior – such contributions are called "extensions". As an effect, the former plug-in may inspect the registered extensions and invoke behavior of plug-ins that are not statically known to the plug-in. Communication is normally controlled by an interface that is associated to a particular extension point, obliging extensions to implement this interface.

3 THE OBJECTTEAMS/JAVA PROGRAMMING LANGUAGE

The Object Teams programming model [7] is realized by the Java-based language ObjectTeams/Java (OT/J) [5]. The language and its tools have matured over a number of years and have been applied in industrial case studies.

Object Teams promotes the notion of collaborations as modules for interacting roles. It does so by introducing two new kinds of modules: *teams* as a higher-order module for contained *roles* (see `MyTeam` and its roles `Role1` and `Role2` in Fig. 1).

A `playedBy` relationship is used for binding a role class to a base class, which will be reflected by run-time links between pairs of role and base instances. The main purpose of creating a role-to-base connection is to create a channel for specific communication, which is established by two kinds of method bindings.

A *callin* method binding intercepts the control flow at a method of the base entity and redirects it to a role method. Looking at Fig. 1, calls to `method2` will be intercepted, causing an invocation of `roleMeth2`. In order to ensure that the effect is purely additive, one of the modifiers `before` or `after` can be used. When specifying a `replace` callin binding, this has approximately the same effect as overriding a method in the context of inheritance. Using the concept of callin bindings, roles unify instance-based inheritance with method call interception as it is used in aspect-oriented programming.

Conversely, a *callout* method binding simply forwards a method (`roleMeth1`) from a role instance to its *player*, the base instance (`method1`). Callout bindings per se are rather unspectacular, but a callout binding may support *decapsulation*, a term we coined for the inverse of encapsulation. This means that a callout binding can – within limits – access protected or even private members of the corresponding base class (note the "holes" in the border of the base package and its class `C2`). Decapsulation is also supported for the `playedBy` clause at class level, so that a role may be attached to an otherwise inaccessible class (like a package-visible class or a private inner class).

Several means exist to control when a callin binding is actually effective, i.e,

[2]For a comparison see http://www.eclipsezone.com/articles/extensions-vs-services/

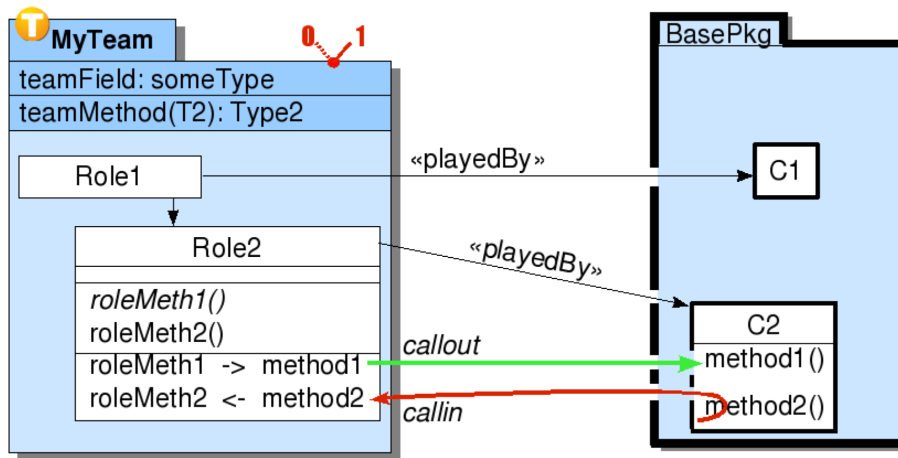


Figure 1: Essential concepts of ObjectTeams/Java

whether or not the base control flow should be intercepted. The most elegant technique is to *activate* or deactivate a given team instance, which has the effect that all callin bindings of all contained roles are enabled or disabled in one step. In Fig. 1, activation is symbolized by the switch at the top of `MyTeam`.

4 A REAL-WORLD EXAMPLE: DEVELOPING THE OTDT

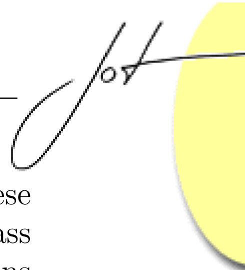
The initial motivation for combining ObjectTeams/Java (OT/J) and Equinox arose during our own development of an IDE for OT/J. To set the stage for OT/Equinox, we will first present some design choices from the development of our tools and explain why we were not satisfied with any of the available alternatives.

Difficulties and standard solutions

The IDE for ObjectTeams/Java is an extended version of Eclipse and its Java Development Tooling (JDT). The resulting IDE is called OTDT (Object Teams Development Tooling[3]). After creating the OT/J compiler by invasive modification of Eclipse's Java compiler, we wanted to extend other parts of the IDE as well. While seeking complete and seemless support for OT/J within Eclipse, we wanted to re-use existing plug-ins, especially the JDT-UI, which implements the user interface for the JDT. Since this task required a number of adaptations of the JDT-UI, we started out by investigating its provided APIs and extension points. It soon became clear that we had to adapt a number of elements that were not intended for adaptation. Similar problems are also reported in [13].

After extension points were ruled out for some tasks, we tried to subclass existing classes from the JDT-UI. In some cases, we succeeded in implementing our adaptations in such subclasses. In other cases the subclasses would have needed to access

[3] Available from `http://www.ObjectTeams.org/distrib/otdt.html`



private members from their superclasses, thus also ruling out subclassing. In these situations we commonly reverted to copy-and-paste re-use, i.e., copying the class carrying the private members into our own plug-in and performing our adaptations invasively on these copies.

Either way, using subclassing or copy-and-paste, we then had to install our classes into the existing plug-in. If no extension point was available, this meant replacing all program locations that instantiate the original class and instantiating our own class instead. As these program locations of instantiation were not generally designed for adaptation either, this usually resulted in a ripple effect of transitively copying several classes merely in order to install one adapted class.

It should be noted that the strategy of transitive copy-and-paste produced an operational prototype which, *looked at from the outside*, was quite satisfactory. Real problems arose when we migrated the OTDT, which was at that time built on Eclipse 3.0, to Eclipse 3.2. The experienced reader will be able to imagine, without further elaboration, the trouble and effort caused by these copy-and-paste classes during migration. This is what made us consider self-application of OT/J for its own tools.

How ObjectTeams/Java can help

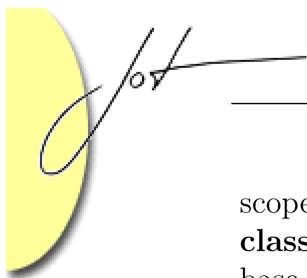
To show how Equinox and OT/J were fruitfully combined, we first identify those features of OT/J that are relevant to the problems mentioned above. The next section describes a way of declaring OT/J adaptations in a manner that is compatible with the component model used.

First of all, we frequently merely wanted to replace the implementation of an existing method with our own implementation or even simply add a few statements at the front or the end of a method. In OT/J, this is supported by **callin** method bindings, whose options of *before*, *after* or *replace* reflecting exactly the desired composition styles.

By simply using one callin binding, some cases that previously involved copying several classes could actually be reduced to a single team of less than 20 lines of code.

When implementing a role method that is supposed to replace a base method, the role method resides within a different class, or even a different plug-in, than the method it replaces. This means that some of the members that need to be accessed from the role method's implementation are not normally visible. To make these needed members visible, **callout bindings** can be used that utilize **decapsulation**. In addition to callout method bindings, OT/J also supports access to fields of a base object, even if regular access control would not permit this. Such **callout-to-field bindings** can be seen as externally defining accessor methods for fields.

Not only methods and fields can pose problems regarding visibility. Some signatures of these members may mention classes that are not visible outside the original



scope. In addition to the member-level decapsulation, OT/J also supports **base class decapsulation**, meaning that a role class can be bound to an inaccessible base class. Note, that base class decapsulation does not simply discard given access restrictions but defines a role class as a gateway to the inaccessible base class.

We were surprised to find that this minimal set of concepts sufficed to solve the majority of issues, where previously we had to revert to copy-and-paste. How we actually made use of the features of Object Teams is described in Section 6. First, however, we will show how the techniques of callin, callout and decapsulation were integrated into Equinox.

5 PUTTING IT TOGETHER: OT/EQUINOX

Combining OT/J and Equinox involved a technical level of connecting run-time infrastructures and a conceptual level of declaring component relations. We start with a minimal understanding of the technical implications, present the high-level concepts and delve into a few technical details towards the end of this section.

New plug-in relationship: aspectBinding

At the conceptual level, integrating OT/J and Equinox involved developing a mechanism for declaring and enforcing a new relationship between plug-ins. How is this relationship mapped to the underlying technology?

Running OT/J applications requires the use of a byte-code transformer that weaves dispatch code into bound base classes. Weaving is performed at class-loading time. The service of aspect weaving is provided by the core plug-in of OT/Equinox: `org.objectteams.otequinox`. Actually, aspect weaving should be seen as a *meta-service* as it does not directly contribute to any application but extends the OSGi framework with a new kind of component relationship: aspect binding. For this purpose, the OT/Equinox plug-in provides the extension point `aspectBindings`, where client plug-ins may register that they want to adapt classes from existing plug-ins. Any plug-in declaring an `aspectBinding` extension is said to be an aspect plug-in. Each extension specifies the following elements:

<table>
<tr><td><code>basePlugin</code></td><td colspan="2">Identifier of a plug-in whose classes may be adapted as base classes.</td></tr>
<tr><td><code>team</code>*</td><td colspan="2">To each base plug-in a number of team classes can be associated with the following subelements:</td></tr>
<tr><td></td><td><code>class</code></td><td>Fully qualified name of the team class</td></tr>
<tr><td></td><td><code>superclass</code></td><td>If the team extends another team from the same plug-in, this is declared here (not used in this paper).</td></tr>
<tr><td></td><td><code>activation</code></td><td>Tells the framework whether the team should be activated automatically.
Possible values are <code>NONE</code>, <code>THREAD</code>, <code>ALL_THREADS</code>.</td></tr>
</table>

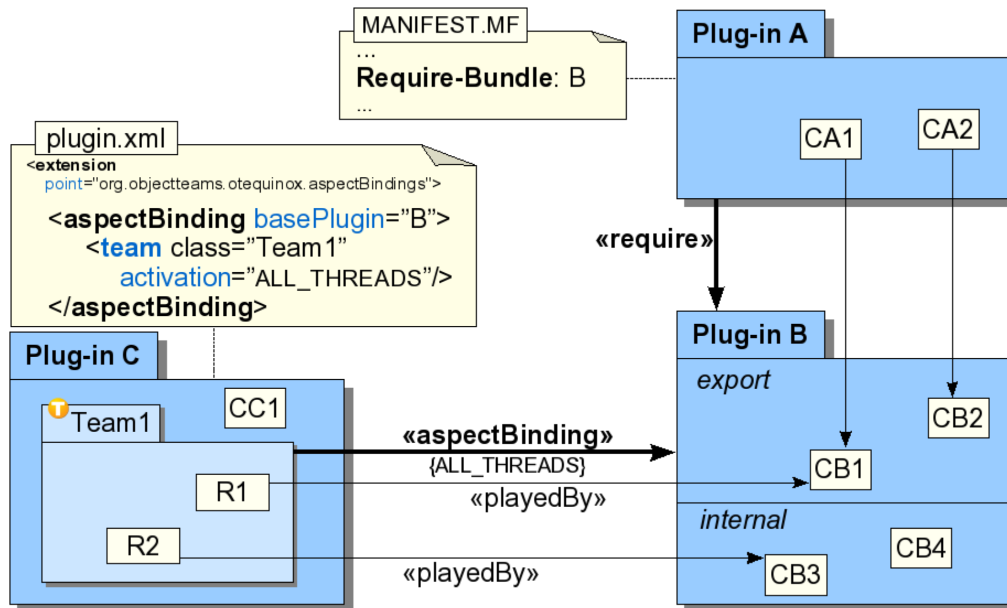


Figure 2: Plug-in relations, including the new *aspectBinding* relation

The effect of an `aspectBinding` extension is as follows (see also Fig. 2): before loading the plug-in specified as `basePlugin` (`B`), the framework loads the given `team` classes (`Team1`) and instantiates each of them. Depending on the specified `activation` mode, the framework activates the team for the current thread, for all threads or not at all. In the latter case, it is the responsibility of the aspect plug-in to activate the team when needed. It is, however, quite common to let the framework automatically activate the team instance.

A plug-in may adapt multiple base plug-ins using `aspectBinding` relationships. Declaring each binding as a relationship between a base plug-in and a list of teams serves two purposes: conceptually this declares that each team in the list may adapt only classes from the given base plug-in. More specifically, any base class in a `playedBy` relation must be a class of the given base plug-in (`CB1`, `CB3` from Plug-in `B`). Technically, the declaration is used to weave any adaptations caused by the given teams into their respective base classes before the base plug-in is fully loaded.

The Figure also shows a traditional component dependency (declared by the `Require-Bundle` directive), which restricts visibility to the exported interface of a bundle. The duality of *require* and *aspect binding* implements the **open-closed principle** for components.

Restricting the loophole

Each `aspectBinding` extension declares an intended loophole between a team class and its base classes in a different plug-in. Through this loophole, method interception and decapsulation can be applied, preparing the ground for a variety of unanticipated adaptations that would otherwise not be possible. Yet, for a modular architecture, it is important to restrict the use of this loophole.

For this purpose, the following rules are enforced:

- The `playedBy` declaration may only be used across plug-in boundaries if a corresponding `aspectBinding` extension is declared.
- Any base class mentioned after the `playedBy` keyword must be imported with the `base` modifier (see below).
- A `base` import restricts the use of the base class within the given team.

More specifically, an import declaration `import base org.prj.internal.AClass;` imports the given class for use in the following program locations only:

- as base class in a `playedBy` binding
- in signatures of base methods as part of a callin or callout binding.

The effect of these rules is that a look at all `base` imports of all teams mentioned in `aspectBindings` gives the full list of classes that are accessed in a special way. Method interception and decapsulation apply to these classes only. On the other hand, these base classes cannot be accessed directly but only via their respective role. If desired, OT/J's advanced concepts of encapsulation (confined roles [8]) can be used to strictly forbid any reference to a role instance from outside the enclosing team instance. Thus, the extent of the loophole is explicit, rather than allowing unlimited tampering with foreign classes.

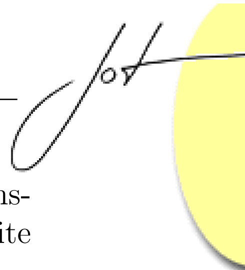
Integration via class loaders

For the technical integration, it is important to know that both technologies, OSGi and OT/J, make essential use of class loaders. In OSGi, each bundle has its specific class loader that is equipped with the dependency information from its bundle manifest. One purpose of this class loader is to ensure that a bundle can only load classes from legal scopes, where scopes are identified with class path entries.

The Object Teams Runtime Environment (OTRE) uses class loaders for a different purpose: by hooking into the class-loading process, the OTRE intercepts the byte code of all base classes to be adapted by any team class. For each base class, the OTRE performs a number of byte-code transformations before actually loading the class into the JVM.

Hooking into the class-loading process may easily interfere with a framework's class-loading strategy. For this reason, we did not use either of the mechanisms we use for standalone OT/J applications[4]. We were fortunate to find a set of hooks in the Equinox framework that were introduced in Eclipse version 3.2 (these hooks are not part of the OSGi standard). Using these hooks, we can easily intercept a number of life-cycle events for each plug-in and can also intercept the class loading proper. Intercepting is implemented by a *fragment* which can be deployed separately, so that the Equinox framework regards it as part of the OSGi itself.

[4] Currently, the normal mode of executing OT/J programs uses the JMangler framework. A variant based on JPLIS technology and Java agents is under development.



Given these mechanisms, resolving dependencies and controlling byte-code transformations during class loading became feasible, but the task turned out to be quite delicate. Two problems had to be solved:

1. The order of loading and instantiating classes must be constructed in a sophisticated way because (a) loading any adapted base class requires loading all adapting teams and their roles in advance because the roles contain the actual weaving instructions, and (b) no team may be activated before all its base classes are loaded because team activation is implemented as registration at all relevant base classes, and (c) no base code should be executed before the adapting teams are activated.

2. The byte-code transformations insert instructions into base classes, which invoke code in adapting teams and roles, thus realizing the required method-call interception. While this means that the woven base class needs to refer to aspect classes, it is, however, not desirable for a base plug-in to explicitly mention this aspect dependency. Even worse, if it listed its aspects, this would result in illegal cyclic plug-in dependencies.

Both problems are solved by OT/Equinox and require no intervention by the developer. The first problem required a carefully crafted state machine. The details of this state machine are beyond the scope of this paper. The second problem is solved by a so-called `BridgeClassLoader`. This special class loader creates a bridge from a base plug-in to all its adapting aspect plug-ins. If a base plug-in fails to load a class by the normal mechanisms, its BridgeClassLoader asks all associated aspect plug-ins for the required class. On the other hand, all of these aspect plug-ins will have a dependency on the base plug-in in question. The BridgeClassLoader must therefore take care not to cause infinite recursion during lookup. By delegating to the aspect plug-in, it is also ensured that both plug-ins use the same instance of the required class (i.e., it is loaded by the same class loader).

At the technical level, the BridgeClassLoader reflects what makes the `aspectBinding` relationship special at the conceptual level: while regular plug-in relationships are unidirectional, the `aspectBinding` relationship is bidirectional. The dependency between the class loaders of base and aspect plug-ins must therefore also be bidirectional.

6 APPLYING OT/EQUINOX

OT/Equinox is used significantly in the implementation of the OTDT itself, as motivated in Section 4. At the time of writing, we have developed five plug-ins defining `aspectBinding` relations. Using a Physical Lines Of Code metric, these plug-ins range from 29 PLOC to about 2500 PLOC. We present some examples from these plug-ins and show what we have learned while developing them.

AboutAdaptor

The first OT/Equinox plug-in was developed to provide transparency about aspect weaving. The so-called `AboutAdaptor` adapts the "About Plug-ins" dialog of Eclipse in order to show which plug-in is adapted by another plug-in. This adaptation is accomplished by one `aspectBinding` stating that team `AboutAdaptor` adapts the plug-in `org.eclipse.ui.workbench`, which implements the "About Plug-ins" dialog. The team contains one role containing one method that intercepts a method of base class `AboutBundleData` using a `replace` *callin* binding. Within the replacing method, one *callout* is used to obtain information from the base object. From that information, a new version string is assembled and returned to the caller. This is perhaps the most simple useful aspect one can think of.

By means of this adaptor, the version number displayed in the dialog is enriched with information about the adapting plug-in. Without OT/Equinox, we would not have been able to provide this information within the existing dialog, *unless* the base plug-in were changed to provide an extension point specific to this scenario.

Adapting the JDT UI

We have developed one plug-in specifically for the purpose of adapting the JDT UI to support OT/J. This plug-in is a collection of 15 teams and 7 regular classes. The simplest of these teams are structurally similar to the `AboutAdaptor`.

A more challenging example is the `CallHierarchyAdaptor`. This team adapts the call hierarchy view, which allows to interactively explore the method call graph of a program. The reason for adapting this view relates to OT/J's method bindings: callout and callin bindings constitute new control flows that could not be displayed by the existing view. Using a single team class of no more than 170 PLOC and containing 7 role classes, we were able to re-use the existing implementation and effectively integrate support for callout and callin method bindings. Roles of the `CallHierarchyAdaptor` intercept 5 methods of the JDT UI. 4 fields and 7 methods are accessed using callout, most of which require decapsulation.

Other teams in this plug-in serve purposes like (incomplete list):

- Vizualizing new program constructs (new icons) and existing Java constructs of which new flavours are defined (decorated with overlay icons). Do this consistently throughout different views.
- Consistently installing OT/J syntax highlighting to views showing source code.
- Extending code assist and rewriting functionality, like (1) creating a callout method binding from a list of available methods, (2) inserting import declaration with or without the `base` modifier depending on the current context, (3) adding and enhancing quick fixes for context-sensitive corrections.
- Toggling between two different ways of displaying so-called role files (inner classes residing in separate files within a designated "team package").

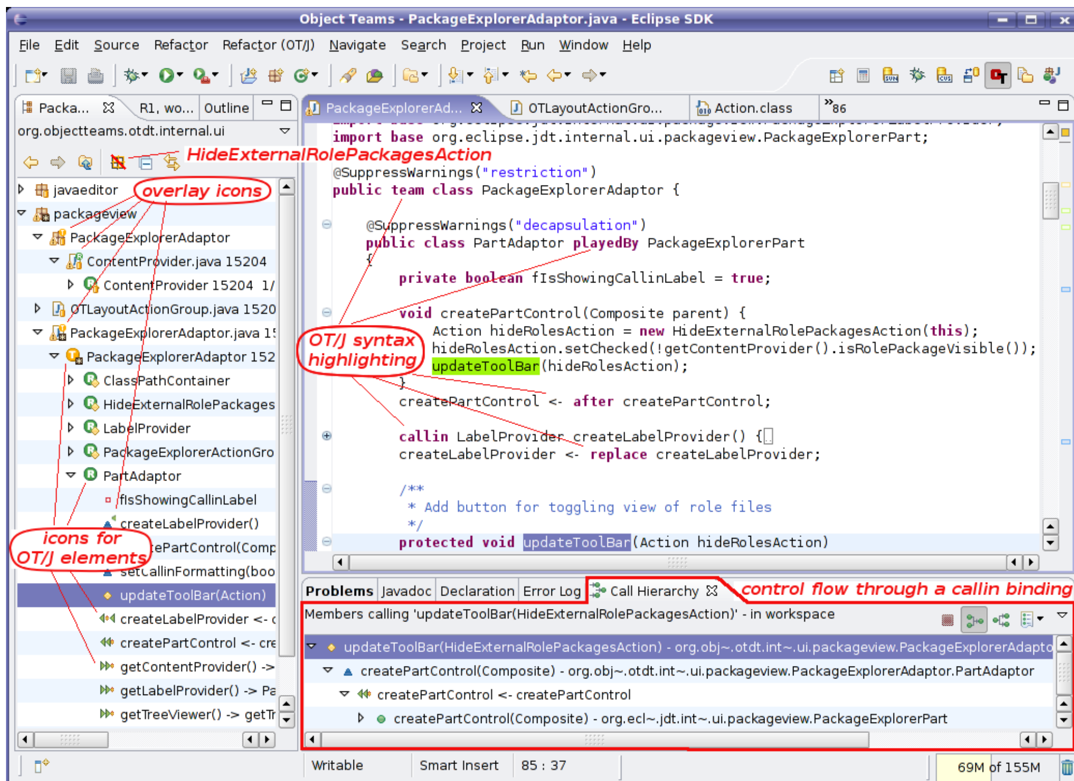


Figure 3: The Object Teams Development Tooling in action.

Figure 3 shows the OTDT in action, illustrating several of the mentioned adaptations. By re-use and adaptation of the existing JDT UI, this aspect plug-in –within less than 2500 PLOC– realizes significantly more functionality than any implementation without aspects would have achieved. Without OT/Equinox, some of these features would only have been possible by copying and modifying large amounts of existing code.

Adapting incremental compilation

The aspect plug-ins mentioned above were motivated by the desire to adapt plug-ins that we do not *own*. Another use-case of the OTDT is also implemented using an aspect plug-in, although we could have chosen to do all adaptations in situ.

When developing the OTDT, we were faced with the following situation: inside the JDT core, the compiler and the incremental builder cooperate to perform minimal compilation after a source file has changed. After extending the compiler for OT/J, the dependency rules for incremental compilation also had to be adapted.

What sounds simple at the requirements level turned out to involve many classes from quite unrelated parts of the JDT core. Any traditional implementation of this requirement would have ended up scattered over large parts of the JDT core. To maintain instead a modular design, we implemented all adjustments of the incremental compilation in a single aspect plug-in, which intercepts control flows from a

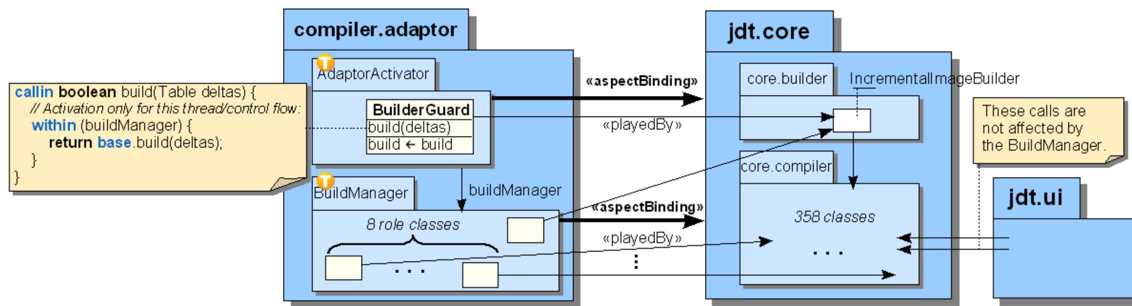


Figure 4: Integrating the compiler adaptor into the JDT.

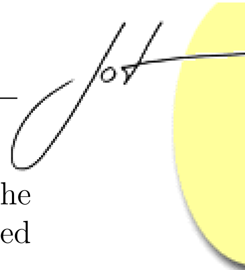
number of different locations within the compiler and the incremental builder (see also the `BuildManager` team and its bindings in Fig. 4).

This plug-in leverages a new design option introduced by OT/J: we use a team instance to mediate information flows between its roles. Thus, when a role observes an interesting event in the base application, it may store some information in the enclosing team. When another role subsequently intercepts the control flow at a point where a new condition needs to be checked, it asks the enclosing team for the information that was stored by the former role. Applying such a mediator style, a team can be used to make information flows explicit, even between entities that have no knowledge of each other. The base classes involved in this scenario live within different modules of the base plug-in. They have no direct communication.

The team `BuildManager`, which applies the mediator style for adapting the compiler, is by far the most complex team within the implementation of the OTDT. Adapting the compiler is further complicated by the fact that the compiler is invoked by many different clients within the JDT. Most clients only use the compiler to construct internal data that is used at the UI for certain purposes. Few of the compiler's clients actually use the compiler to generate Java byte code, and only the incremental builder requires the compiler to recursively trigger compilation of dependent classes.

Thus, our adaptation of the compiler, which in specific situations needs to communicate with the incremental builder, must not assume that the incremental builder is actually active while the compiler operates. This led to the cascaded aspect design shown in Figure 4.

This design involves two teams: an `AdaptorActivator`, which is globally activated via the `aspectBindings` extension (not shown), and a `BuildManager`, which is inactive by default. Only when method `build` is invoked on the base class `IncrementalImageBuilder` is this method call intercepted by the role `BuilderGuard` (callin binding `build <- build`). The intercepting method (also called `build`, shown in the note) invokes the original behavior (`base.build(deltas)`), but now the build method is called from a `within` block, which means that it is called within the context of a locally activated team `buildManager`. Here, locally means that the team is activated only *for the current thread* and only until the `within` block finishes (nor-



mally or by an exception). Using this single mechanism, *all* callin bindings of the team `BuildManager` (there are currently 10 such bindings) are consistently enabled if and only if the compiler is invoked by the incremental builder.

7 LESSONS LEARNED FROM THESE EXAMPLES

The examples have shown several uses of teams as an advanced modularization and adaptation mechanism.

Role containment. The `BuildManager` is a module for a set of contained roles. Containment can even be *nested*, which means that a team can contain teams at arbitrary depth, thus allowing for recursive scalability inside a plug-in.

Programming with views. The same team `BuildManager` also demonstrates the use of teams and roles as views: by attaching roles to all relevant base classes, a team defines a selective view of the base plug-in, bringing closely together some elements that at the base level may be at any distance from each other. More specifically, 8 roles of the `BuildManager` constitute a subjective abstraction of the 358 classes of the compiler. Such a design has the advantage of keeping together in one single module all those pieces of code that contribute to the same requirement, here: adjusting incremental compilation. The concept of views includes the option that other plug-ins may have different views of the same base plug-in. Roles of different views (teams) do not interact with each other, except for indirect communication via the shared base.

The OTDT also contains one aspect plug-in that needs to adapt *several* base plug-ins. In Section 8, we will discuss approaches that actually allow one single aspect class to adapt classes from any base component. When designing the `aspectBindings` extension point, we decided against such freedom. To support the use of an aspect binding as an abstraction over a well-defined set of lower-level dependencies each team can only adapt base classes from one base plug-in. However, an aspect plug-in may, of course, contain several teams, where each team may adapt another base plug-in. Using each team as a view for its base plug-in, an aspect plug-in may implement interactions between these teams and thus coordinate the behavior of several base plug-ins. In the same way as roles provide selective views of classes, teams may thus provide selective views of other plug-ins in the system.

Support for views helps to develop small modules with good cohesion for concerns that would otherwise be implemented over a large set of distant classes (*scattering*).

Leveraging thread context. Finally, teams can also be used to reify thread contexts. Using the example of the `AdaptorActivator`, we have shown how per-thread team activation can be used to let the same base plug-in behave differently depending on the current thread context and control flow.

Evaluation

Our examples show that OT/Equinox actually helps to reconcile the two primary goals of scalability and adaptability, while preserving the added value of OSGi.

Scalability is obtained by both the bundle concept of OSGi and the composition features of OT/J. All the aspect plug-ins we developed are actual plug-ins with well-defined interfaces to other plug-ins. In addition to normal imports/exports, we use one additional relationship: `aspectBinding`[5]. This relationship is a coarse-grained relationship between plug-ins (which is declared on the aspect side only), supporting the building of systems from components (plug-ins). While the absence of any aspectBinding relationship guarantees that no adaptation between plug-ins takes place, it is possible to zoom into each aspectBinding to find out:

- Which teams adapt a given base plug-in?
  (This information is in the plug-in descriptor.)
- Which base classes are adapted by a given team?
  (This information is in the team's base import list.)
- To which methods and fields an aspect has special access?
  (This information is in the callout/callin bindings of the role adapting the given base class.)

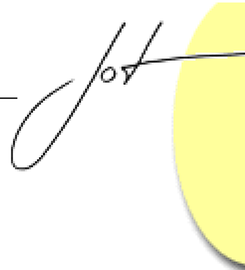
The above list is complete for any adaptation of a given base class. Thus, aspect bindings are a component-level relation that abstracts from the lower levels of classes and methods.

Within each aspect plug-in, teams also provide new design options for scalable architectures. Not only can a team be used to effectively encapsulate a set of roles, team activation also helps to raise the level of abstraction from individual message flows to complex behaviors that can be turned on and off by a single switch.

Adaptability is obtained by the very concepts of OT/J: the playedBy relation and its refinements, callin and callout. Given these fundamental options, the development of our aspect plug-ins for the OTDT demonstrates that re-use of the JDT UI plug-in *is* possible, even if requirements significantly deviate from the original ones. Most of our requirements could not be integrated into the JDT UI by using provided extension points because our adaptation scenarios were not considered by the JDT UI developers. Given that the JDT UI is a plattform for a large number of tools, the developers of the JDT UI are actually right in not cluttering their design with new extension points that would hardly serve anybody besides us.

Added Value. Since any aspect plug-in is a regular plug-in, it benefits from the added value of OSGi. Aspect plug-ins are packaged and deployed using standard mechanisms. Life-cycle support also includes aspect plug-ins, e.g., with respect to on-demand loading. The semantics of plug-in activation is seamlessly extended with new options to automatically activate teams.

[5] Remember that, technically, `aspectBinding` is an extension point, whereas, conceptually, this meta-service actually adds a new kind of relationship between plug-ins.



8 RELATED WORK

The idea of combining the strength of mature component models with the flexibility of aspect-oriented programming languages has motivated several integration efforts.

In programming language design, the proposal of **Aspectual Components** [14] has inspired the development of several approaches including OT/J. Most of the language features that were important for developing OT/Equinox actually relate to the original Aspectual Components. This means that many of our findings could potentially be generalized to other programming languages in this group. Decapsulation, however, is not part of that model.

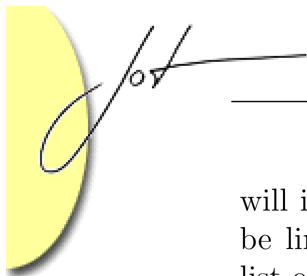
Among this group of approaches, **JAsCo** [21] comes closest to the work presented in this paper. JAsCo is an aspect-oriented language extension to Java that defines its own component model, where components are regular classes or *Aspect Beans*. Aspect Beans are a means to group *hooks* that adapt other components, while a separate connector defines the bindings between them. Still, the component model of JAsCo is not as ambitious as the OSGi, so JAsCo is strong in flexibility but its scalability remains to be shown.

None of these language-level approaches has a relevant focus on added values (with the exception of distribution support in JAsCo). Looking from the component side, Spring and JBoss both have extensions relevant for this discussion.

Spring AOP [11] integrates limited AOP functionality with the light-weight component model of the Spring framework. In this model, Spring *beans* are fine-grained components (Java classes) that can be configured from the outside through dependency injection and that are managed by the Spring container. Aspects are implemented as regular Spring beans, while aspect-specific details like pointcuts or aspect declarations can be defined inline via annotations or in a separate Spring configuration file. With Spring AOP, it is only possible to intercept and advise method invocations on Spring-managed beans. To overcome this and other limitations, one can also use AspectJ [1] with Spring, which offers more AOP features but is unaware of the Spring context (and vice versa). Spring's component model lacks any concept of modules larger than classes (not counting the so-called "contexts", a rather weak concept that does not provide encapsulation).

JBoss AOP [12] is an AOP framework for Java, which is integrated with the JBoss application server to support user-defined aspects for Java Enterprise (JEE) applications. Here, the component model is that of JEE (e.g. Enterprise Java Beans) and the AOP features are comparable to those of AspectJ. JBoss AOP uses plain Java and a combination of annotations and XML configuration files to define aspects. As a result, it is possible to use AOP inside the JBoss application server, but those aspects can neither be components nor do they respect the boundaries of components, i.e. aspects undermine the explicit component structure of a system.

In the context of Equinox, **AOSGi** [16] is being developed as an integration of AspectJ [1] into Equinox. In contrast to the related approach AJEER [15], AOSGi



will indeed support component-level declarations, enabling the scope of aspects to be limited: they introduce an `Eclipse-SupplementBundle` directive indicating a list of bundles that can be adapted by aspects in the current bundle. More specific control on the aspect-base interface, comparable to our efforts on "restricting the loophole", is not possible. AspectJ's privileged aspects, for example, may break the encapsulation of *any* class in the system. Furthermore, aspect modules in AspectJ are a derivation of classes, which are, however, not perfectly integrated with polymorphism [3]. OT/J, on the other hand, is much better prepared for scalable architectures due to the team module concept.

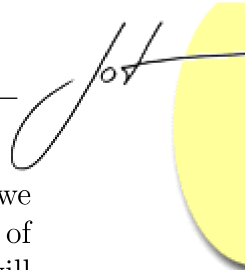
On the road towards combining scalability, adaptability and added value, existing approaches are strong on one or two of these goals but fall short on another goal. Those that extend a programming language with a component model usually provide little added value. On the other hand, extensions of existing component frameworks have difficulty defining aspect binding in such a way that it does not undermine the clear component structure of a system.

9 CONCLUSION AND FUTURE WORK

We have presented OT/Equinox as an integration of the programming language ObjectTeams/Java with the OSGi component model as implemented by the Equinox framework. We have shown that the resulting technology indeed combines the best of both worlds, resulting in full support for scalability and adaptability while preserving the added values of OSGi. A key feature of our approach is the fact that aspect bindings are defined as first-class component relations. We are aware of the fact that undisciplined use of decapsulation and interception (callin) *can* result in unmanageable designs. Because avoiding aspect technology would in many cases result in the prohibition of re-use, which we consider unacceptable, we are seeking to make the aspect binding relation as safe as possible. Rules for the disciplined used of aspect bindings are therefore enforced in order to support explicit guarantees about the effect of aspects even at the component level. In order to further strengthen the contracts between aspects and bases, we are currently integrating mechanisms for *confirmed join points* [20].

We have shown that both individual technologies are well suited for integration. Actually, teams support some styles of composition that may even scale better than regular component composition: such styles are nesting, stacking and layering [9]. This significantly improves the approach's scalability and also inspires future considerations about, for example, the nesting of plug-ins. Equinox, on the other hand, is flexible enough to support our extension by using hooks regarding the bundle life cycle and the setup of class loaders. Also, the concept of extension points was essential for defining the new component relationship.

We have presented examples from a real-world application, where OT/Equinox enabled the re-use of existing plug-ins with considerably changed requirements. This



re-use would not have been possible with regular technology. In the near future, we will perform the next step of **evolution** when (again) migrating to a new version of Eclipse. We are confident that, compared to previous migrations, this next step will demonstrate that our aspect plug-ins cause much less effort during the migration than the previous copy-and-paste strategy did.

Based on communication with other groups working on extensions of the Java language, we see great potential for also minimizing their efforts to build full-fledged IDEs by adapting the JDT. We have also used OT/Equinox for an adaptation of the Eclipse Visual Editor and are currently planning the application of OT/Equinox for the development of domain-specific graphical editors based on the Eclipse Graphical Modeling Framework and UML2.

Eclipse makes only limited use of OSGi's capability for dynamic loading/unloading of bundles. Future work on OT/Equinox will strive for supporting fully dynamic aspect bundles, too. This includes the capability to dynamically weave, un-weave and re-weave aspects at run-time.

By the combination of bundles and aspects, OT/Equinox successfully raises the open-closed principle from classes to components.

REFERENCES

[1] The AspectJ Team. The AspectJ^{TM} Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide.

[2] Uwe Assmann. *Invasive Software Composition.* Springer Verlag, Berlin, 2003.

[3] Erik Ernst and David H. Lorenz. Aspects and polymorphism in AspectJ. In *Proc. AOSD 2003*, pages 150–157, Boston, USA, 2003. ACM Press.

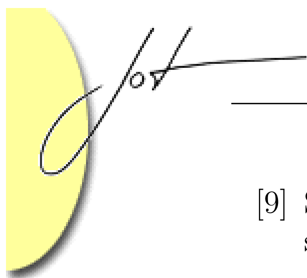
[4] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[5] S. Herrmann, C. Hundt, and M. Mosconi. ObjectTeams/Java language definition. Technical Report 2007/03, Technical University Berlin, 2007.

[6] S. Herrmann, M. Mezini, and K. Ostermann. Joint efforts to dispel an approaching modularity crisis. divide et impera, quo vadis? In *Proc. of 6th International Workshop on Component-Oriented Programming (WCOP)*, 2001.

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

[8] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin, 2004.



[9] Stephan Herrmann. Sustainable architectures by combining flexibility and strictness in Object Teams. *IEE Software*, 151(2):57–66, April 2004.

[10] Stephan Herrmann. Programming with roles in ObjectTeams/Java. In *Proc. AAAI Fall Symposium "Roles, An Interdisciplinary Perspective"*, Arlington, USA, November 2005.

[11] Interface21. Spring Framework home page. http://www.springframework.org.

[12] JBoss.org. JBoss AOP home page . http://labs.jboss.com/portal/jbossaop/.

[13] M. Kersten, M. Chapman, A. Clement, and A. Coyler. Lessons learned building tool support for AspectJ. In *"AOSD 2006 - Industry Track Proceedings". Technical Report IAI-TR-2006-3, ISSN 0944-8535*, Computer Science Department III, University of Bonn, 2006.

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

[15] Martin Lippert. AJEER: an aspectj-enabled eclipse runtime. Demonstration at OOPSLA 2004.

[16] M. Webster M. Lippert. equinox incubator . http://www.eclipse.org/equinox/incubator/aspects/index.php.

[17] Bertrand Meyer. *Object oriented software construction*. Prentice Hall International, New York, second edition, 1997.

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

[19] OSGi service platform, r4 core v4.0.1 specification. Technical report, The OSGi Alliance, http://www.osgi.org/osgi_technology/download_specs.asp, 2006.

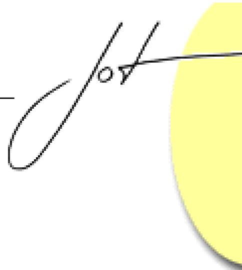
[20] Harold Ossher. Confirmed join points. In *Proc. of the SPLAT workshop at AOSD'06*, Bonn, Germany, 2006.

[21] D. Suvee, W Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proc. AOSD 2003*, pages 21–29, Boston, USA, 2003. ACM Press.

[22] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[23] Peri Tarr, Harold Ossher, Stanley M. Sutton, Jr., and William Harrison. N degrees of separation: Multi-dimensional separation of concerns. In Filman et al. [4], pages 37–61.

[24] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering. Addison-Wesley, 2001.



ABOUT THE AUTHORS

Stephan Herrmann received his Ph.D. at Technische Universität Berlin in 2002 for his work on applying new techniques for separation of concerns to the development of a multi-view software engineering environment. Since then his focus is on developing the language ObjectTeams/Java and its tools. He has taught ObjectTeams/Java both at university and in tutorials at international conferences.

Marco Mosconi is a Ph.D. student at Technische Universität Berlin, working in the field of model-driven and aspect-oriented software development. He has a strong focus on component-oriented target platforms and did already work on the integration of aspect-oriented concepts and component models as part of his diploma thesis.