Eclipse plugin adaptation with Equinox and ObjectTeams/Java

Stephan Herrmann Christine Hundt* Carsten Pfeiffer*
Technische Universität Berlin Fraunhofer FIRST
{stephan,resix}@cs.tu-berlin.de carsten.pfeiffer@first.fhg.de

ABSTRACT

Although Eclipse provides powerful mechanisms for extension this flexibility is constrained by depending on the foresight of each plugin's developers: extension points have to anticipate future extensions, access restrictions aim for encapsulation but sometimes sacrifice adaptability, and the rule of "adding, not replacing" imposes difficulties on some forms of reuse. As a result, some Eclipse-based projects take the cumbersome road of duplicating significant amounts of plugin-code, just in order to insert their specific adaptations, leading to maintenance nightmares abound. This position paper presents the aspect-oriented programming language ObjectTeams/Java as an alternative for smoothly adapting existing plugins. We have developed a small plugin that utilizes the new hooks in the OSGI framework in order to support load-time aspect weaving for Eclipse plugins written in ObjectTeams/Java. While this work builds upon some experience from previous works regarding AspectJ [7] we briefly discuss why we believe that the use of ObjectTeams/Java for Eclipse plugins should lead to even cleaner architectures.

1. REUSE CONFLICTS

The Eclipse plugin concept supports different scenarios for software composition and re-use which are difficult to classify using traditional concepts such as modules, components or frameworks. Some combinations of plugins interoperate in a style that strictly adheres to a black-box component model. Other plugins should rather be classified as frameworks where substantial white box knowledge is used to create specialized versions of the original plugin. Ideally, such specialization is performed using extension points and public API only. In real life it frequently occurs that perhaps just one extension point is missing for a required adaptation, or a class or member lacks the necessary permissions to access or override. This problem can not be blamed on the developers of reusable plugins, but is inherent to the situation of framework development in general. The roles of framework developer and application developer are inherently in conflict: The framework developer wants to protect his or her software by using encapsulation and only explicitly exposing specific adaptation points. The application developer wants to have license to view and adapt each single piece of the framework using sub-classing and overriding.

This conflict can further be sub-divided into two questions:

1. Do adaptation points have to be declared explicitly or should each element in a framework be considered a potential adaptation point?
2. To which extent is encapsulation a discipline that leads to correct and robust software, and at what point does encapsulation turn against developers preventing them from achieving their tasks?

Regarding question (1) we assume, that no framework developer ever can anticipate all future adaptations to be performed on his or her framework. Even if this were possible the resulting design would be bloated with myriads of specific configurable hooks that in most cases will never be used at all. This is (in part) a technical question that can be solved by providing more flexible means of adaptation. We argue that on-demand adaptation points allow for a wide range of valuable compromises between pre-planning and ad-hoc adaptation.

Also question (2) can be mitigated by more flexible protection rules. However, the core of this conflict is a social conflict: one developer owns a piece of software and wants to protect this from other developers fooling around with this piece, because this developer wants to protect him/herself from being blamed for bugs that are produced by others. More openness furthermore means additional maintenance efforts because each detail, once publically exposed, must be preserved during evolution. Application developers on the other hand frequently only have these two options: (a) breaking encapsulation (e.g., by copying large amounts of source code and modifying it in-place) *or* (b) just not using the given framework, because it cannot be adapted such as to fulfil the new requirements.

In this paper we focus on technical support for new kinds of modular designs for adaptable plugins. In the end we will briefly discuss the social implications and how this technology can be embedded into new forms of trading.

2. PLUGIN DESIGN WITH OBJECT TEAMS

The current Eclipse concept for plugin extension is *anticipated* adaptation via predefined extension points. Aspect-oriented concepts generalize this extension mechanism by the introduction of joinpoints which can bee seen as *unanticipated* extension points. Further with aspect-oriented concepts it is possible not only to add new behavior but also to replace functionality of a plugin. With aspect-oriented

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

design principles it is thus possible to express new kinds of relationships between Eclipse plugins.

In this sense the aspect-oriented programming model Object Teams [1, 2] offers two new design options that help mitigating the above re-use conflict: *roles* and controlled *decapsulation*.

2.1 Programming with roles

Using Object Teams, aspects are constructed from special classes called roles. A role class has the option to declare a base class by which this role is played (using the keyword `playedBy`). A role class can adapt the behavior of its base class much like a sub-class can do with respect to its superclass. The main difference between sub-classing and roles lies in the fact that roles are kept as separate entities at runtime. By this concept instances of a base class being adapted are regular instances of that very class. Only by the addition of a role instance a base instance will be adapted. This means that roles can be added and removed at runtime and that any number of roles may co-exist adapting the same base instance.

This technique combines three valuable properties:

- Base instances are kept intact exactly as they were originally implemented.
- Roles can be added independently from each other (of course semantic interaction via the base instance may have to be considered).
- The effect that roles have on their base entities may change over time.

Roles are composed to so-called *teams*. Role classes can be regarded as inner classes of team classes, each role instance lives in the context of an enclosing team instance. The team adds the following properties:

- A team creates roles on-demand whenever adaptation of a base instance is needed for which no role exists yet.
- A team instance can be activated or deactivated at runtime which controls whether the behavior adaptation of all its contained roles is effective or ineffective.
- Teams automatically maintain consistent graphs of collaborating role instances as views of corresponding graphs of base instances. This allows to simultaneously adapt a number of base classes in a consistent way. Individual roles of such a team may safely depend on sibling roles as adaptors for (indirectly) related base classes.

2.2 Controlled "decapsulation"

Object Teams coins the notion of *decapsulation* as the inverse of encapsulation. This means that a role class may access features of its base class even if the normal rules of encapsulation would prohibit such access. In contrast to, e.g., AspectJ's technique of privileged aspects, decapsulation is reported for each individual feature as a configurable compile-time error or warning. We consider developer's awareness of the use of decapsulation as important. This mechanism requires new rules of conduct for developers to use the new power in a responsible fashion.

2.3 Infrastructure for ObjectTeams/Java

The core tooling for the programming language ObjectTeams/Java [4] consists of a compiler[1] which compiles regular Java classes as well as the aspect code (teams and roles) and of a runtime environment (OTRE) component responsible for aspect weaving [5].

The OTRE comprises a set of class transformers, which analyze and transform the byte code of Java classes in order to achieve the weaving of aspects. This includes the insertion of calls to the aspect code into base classes and the definition of the precise effects of dynamic team activation.

The point in time at which these transformations have to take place is the start of an application. Especially in the case of plugin adaptation this is essential, because the affected base classes are only available via the plugin jar files and not necessarily as source code.

The first version of the OTRE uses the loadtime transformation framework JMangler [6]. JMangler aims at classloader independence, hooking into the classloading mechanism and enabling the transformation of classes loaded by every classloader except the bootstrap classloader (system classes).

We are currently developing a version that utilizes the Java Programming Language Instrumentation Services (JPLIS) API and agent mechanism of Java5. Because of the built-in capability of JPLIS this approach depends even less on specific classloaders. It even allows the adaptation of system classes except for those that are needed by the transformer component itself.

3. ADDING OBJECT TEAMS SUPPORT TO EQUINOX

We have developed a prototype for adaptation of eclipse plugins using Equinox. This prototype initiates the transformation process by calling the transformation methods of the OTRE. In this case we do not have to worry about the classloader independence, because Equinox itself is responsible for handling the eclipse specific classloaders.

3.1 Hooking into OSGI

Starting with Milestone 5 of Eclipse 3.2, the OSGI framework supports so-called extension bundles that may hook into OSGI's functionality. For example, there are hooks for byte code transformations as used for load-time aspect-weavers. Our extension bundle provides implementations of the new interfaces `ClassLoadingHook` and `BundleWatcher`. The first interface mainly provides the following hook function:

```
byte[] processClass(String name, byte[] classbytes, ..);
```

[1]In fact this compiler is an adapted version of Eclipse's Java compiler from the JDT core.

By this hook we can easily dispatch to the byte code transformers of the OTRE.

The other interface `BundleWatcher` allows us to observe the status of all bundles while they are loaded. This hook manages the interdependent life-cycle of aspects and the plugins which they adapt. By this the following sequence of actions can be established.

1. A transformer plugin that controls the actual class loading is guaranteed to be activated early during Eclipse's bootstrapping.
2. For each plugin to be installed it is checked whether any known aspect declares to adapt this plugin. Before an adapted plugin is activated the following steps are performed:
   (a) The adapting team classes are loaded. From these classes the OTRE receives its instructions to perform aspect weaving.
   (b) The base classes adapted by the roles of each loaded team are loaded, too.
   (c) The adapting team classes are instantiated and the team instance is activated, to enable the adaptation.

Given the new hooks in the OSGI, developing our transformer plugin was a straight-forward task, except for the handling of different classloaders and their interdependencies. We have to cope with four kinds of universes realized by their specific class loaders:

1. **OSGI Framework**
   A bootstrap class is installed and activated by the new commandline argument `-Dosgi.hook.configurators.include=HookConfigurator`. The hook configurator installs the `TransformerHook` implementing the interfaces `ClassLoadingHook` and `BundleWatcher` as described above. Thus, `TransformerHook` shares the initial classloader of the OSGI framework itself.
2. **Transformer Plugin**
   The plugin `org.objectteams.eclipse.transformer` defines an extension point for registering aspects. This plugin implements the actual loading of team classes and base classes in the order described above.
3. **Aspect Plugin**
   Each aspect plugin defines one or more team classes that are to be loaded in this plugin's classloader. Aspect plugins declare in their plugin dependencies which plugins are needed because they contain base classes to be adapted by the aspect.
4. **Base Plugin**
   Of course also the adapted base plugin has its own classloader.

The tricky point about this setup is a cyclic dependency between classloaders: the aspect's classloader (3) needs to delegate to the classloaders responsible for the base class to be adapted (4). During byte code transformation the OTRE adds into these base classes code that in turn depends on the aspect's classes. Because this cyclic dependency cannot directly be mapped to plugin dependencies (which must not contain cycles) we had to develop a special `BridgeClassLoader`. This classloader allows bi-directional sharing of classes between two classloaders.

Declared plugin dependencies already support sharing of base classes. The opposite direction of sharing is established by the bridge classloader. This classloader is associated with a base plugin but also knows about the aspect plugins adapting the given base plugin. Any class that cannot be found by the base plugin's classloader will be searched using any of the adapting aspects' classloader. Care must be taken not to create infinite recursion if lookup in all these classloaders fails, because each side may delegate to the other side to find unknown classes.

3.2 Declaring Aspect Plugins

As mentioned above, we defined a new extension point for registering plugins containing team classes. A plugin providing an extension to this extension point must list the names of team classes to be loaded by the above mechanism. For each team class it must also be declared which other plugin it adapts. This is all a plugin developer must specify regarding the relationship between an aspect plugin and those plugins it wishes to adapt. At loadtime, the OTRE will derive the set of adapted base classes from the role classes. That information is then used to control the loading process.

This strategy differs from how standalone programs in ObjectTeams/Java are configured. Generally, ObjectTeams/Java does not require any configuration files. A team has only effect in a program if an instance is explicitly created and activated. In order to support addition of aspects without recompilation an ObjectTeams/Java program can optionally be launched with a list of teams to be instantiated and activated. This list is passed via a small configuration file mentioned on the command line.

For aspects that should affect classes from another plugin a more explicit management is desirable to raise visibility of the adaptations incurred by an aspect. Using an extension point for aspect declaration provides the necessary information without requiring any change to the plugins affected by the aspect.

In order to provide this information also to the Eclipse user we adapted the "about plugins" dialog using an aspect. First we made the unfortunate experience, that class `AboutPluginsDialog` poses a quite unforeseen restriction inherited from jface's `Dialog`: class `Dialog` must be loaded with a valid `Display` set, because the static initializer of this class tries to read the `ImageRegistry` of the current display.[2] Since bundle loading does not happen in a valid UI-thread, currently no subclass of `Dialog` can be loaded using this mechanism.

As an intermediate solution we chose to adapt the `AboutBundleData` instead, which was again easy.

[2]See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=142299`. It has been acknowledged that this design is not optimal.

4. SUMMARY AND FUTURE WORK

We have shown the successful combination of two powerful technologies: the plugin concept of Equinox with its new hooks and the language ObjectTeams/Java. By this combination it is now possible to let plugins extend and/or adapt each other in far more flexible ways. Object Teams encourages a developer to think in layers of collaborating roles. This helps to find plugin designs which don't just abuse the new flexibility for "patching" places that are otherwise inaccessible. Instead, a design of plugins written in ObjectTeams/Java should exhibit an excellent structure both internally as well as between interacting plugins. Team classes are modules larger than plain classes, which support different kinds of composition including inheritance. Pushing this idea even further we can imagine to use the concepts of Object Teams in order to define some kind of inheritance even between plugins.

The price for the new flexibility is new flexibility, which is to say: some rules which were strict before, can be ignored using ObjectTeams/Java. As mentioned in the introduction this might be scary for original plugin developers. Our language already supports a means against unwanted decapsulation: if a jar has sealed its packages, the encapsulation of its classes can not be broken by decapsulation [3]. If anxious developers would now start to seal all their packages in all jars they produce, nothing would be gained.

To raise the problem to the appropriate level, we are planning to support a mechanism like *confirmed joinpoints* [8]. The idea is, that a class owner might confirm certain adaptations at certain points of his code, where these allowed adaptations can also be filtered by who is doing the adaptation. This way a plugin could acknowledge a specific other plugin allowing that plugin access to some joinpoints. Such confirmation would not affect any other client. Based on this vocabulary a number of scenarios, migration paths etc. can be thought of, along which developers with conflicting interests can turn towards selective, explicit cooperation. Perhaps, in commercial software development confirming a joinpoint could cost a fee. We are convinced that many developers would rather pay this fee than having to choose between ugly, non-maintainable workarounds or re-coding the whole thing anew from scratch.

Aside from improving the plugin presented in this paper we are planning to apply this technology for a number of plugins thus evaluating the benefit of using ObjectTeams/Java for the development of Eclipse plugins.

5. REFERENCES

[1] S. Herrmann and C. Hundt. ObjectTeams/Java Language Definition version 0.8 (OTJLD). http://www.ObjectTeams.org/def/0.8/, 2002–2005.

[2] 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.

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

[4] Stephan Herrmann, Carsten Pfeiffer, and Jan Wloka. Aspect composition with ObjectTeams/Java in eclipse. Demonstration at AOSD'05, 2005.

[5] Christine Hundt. Bytecode-transformation zur laufzeitunterstützung von aspekt-orientierter modularisierung mit object-teams/java (in german). Master's thesis, Technische Universität Berlin, January 2003.

[6] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler—A framework for load-time transformation of Java class files. In *First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, November 2001.

[7] M. Lippert. AJEER: An aspectj-enabled eclipse runtime. Poster and Demonstration at OOPSLA'04, October 2004.

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