Modular Hot Spots: A Pattern Language for Developing High-Level Framework Reuse Interfaces using Aspects

André L. Santos[1] and Kai Koskimies[2]

[1] Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa
PORTUGAL
andre.santos@di.fc.ul.pt

[2] Institute of Software Systems
Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere
FINLAND
kai.koskimies@tut.fi

Abstract. Applications based on an object-oriented framework can be built by programming against the framework's reuse interface. Mastering a framework is typically a time-consuming and difficult task. This paper presents a pattern language for developing higher level reuse interfaces for an existing framework. When applying the patterns that constitute the language it is implied that the framework becomes enhanced with an additional layer of reusable modules that rely on aspect-oriented programming. These modules are referred to as MODULAR HOT SPOTS. They modularize existing hot spots, enabling a framework-based application to be built in a stepwise way and at a higher abstraction level than if using the conventional reuse interface. By raising the abstraction level, it is intended that the development of framework-based applications becomes facilitated.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schmmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>. Copyright ©2009 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

1 Introduction

An *object-oriented framework* [4] (hereinafter, simply *framework*) embodies the abstract design and implementation of a family of related applications. Framework-based applications are developed against the framework's *reuse interface*, i.e. the classes, interfaces, and methods, which an application developer has to deal with in order to build an application. Depending on the framework nature, an application may be developed by *specialization* (*white-box* reuse) or *polymorphic composition* (*black-box* reuse). Most often, an application has to be developed using both means, given that most frameworks have a *gray-box* nature.

A *hot spot* is a fragment of the reuse interface that enables the adaptation of a certain variation point in framework-based applications. A hot spot typically involves more than one framework class, while on the other hand, a same framework class may be involved in more than one hot spot. Therefore, there is a many-to-many mapping between variation points and framework classes that support them (illustrated in Figure 1).

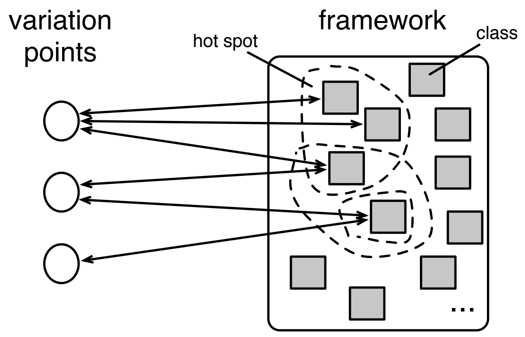


Fig. 1. Many-to-many mapping between variation points and framework classes.

The described many-to-many mapping implies that there are classes of a framework-based application that will be *tangled* with respect to the adaptation different variation points, while the adaptation of certain variation points is going to be *scattered* among more than one class of the framework-based application. *Tangling* implies that modifying the adaptation of a variation point in a framework-based application requires to cope with code statements that pertain to other variation points, whereas *scattering* implies that the modification may involve more than one class.

The work in [8] proposes a technique based on *aspect-oriented programming* (AOP) [5] for developing framework reuse interfaces using *spe-*

cialization aspects (see Figure 2). These are reusable modules that modularize hot spots and enable to build framework-based applications on a higher abstraction level than if using a conventional reuse interface. A framework-based application is implemented in *application aspects* which inherit from the specialization aspects.

This paper presents design patterns for enhancing a conventional reuse interface with specialization aspects, in the form of a pattern language that we refer to as MODULAR HOT SPOTS. The patterns can be used together for solving the problem of developing the several modules which form the higher level reuse interface. By having MODULAR HOT SPOTS it is intended that the new reuse interface has:

– *Modular reuse interface.* Framework-based applications can be developed in a stepwise way. Each application module is the adaptation of a MODULAR HOT SPOT and implements an increment that does not require modifications or knowledge about the internals of existing modules. The capability of developing application increments without having to modify or understand existing code is beneficial with respect to evolution.

– *Less hook methods.* Through the adaptation of MODULAR HOT SPOTS, framework-based applications can be developed without dealing with as many hook methods as in conventional solutions. This contributes to have a *narrow inheritance interface*, a principle that states that only a few hook methods should be required to be given per each ap-

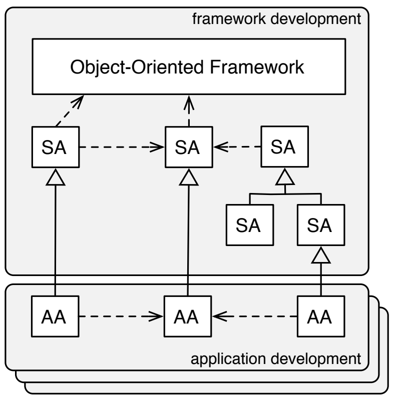


Fig. 2. Specialization aspects (SAs) and application aspects (AAs).

plication class [11].

- *Less application-relevant methods.* Framework-based applications are able to be built without using as many framework methods as in conventional solutions. This may imply that whole framework classes/interfaces will become irrelevant to applications when having the Modular Hot Spots. Having a reduction in the number of framework elements that an application developer has to deal with, reduces the size of the reuse interface, and therefore, facilitates the task of learning it.

Given the above points, the abstraction level is raised with respect to the development of framework-based applications. Frameworks tend to evolve from white-box to black-box [7]. When developing Modular Hot Spots, a framework is transformed in this direction, too. However, the framework reuse interface can become more high-level than the one of a conventional black-box framework.

The knowledge embodied in the language results from developing Modular Hot Spots using AspectJ [1], for the frameworks JHotDraw [9] and Eclipse Rich Client Platform (RCP) [6]. The former is a framework for building editors for structured graphics, while the latter is a framework for building GUI applications based on the Eclipse's dynamic plugin model and UI facilities.

The target audience of this pattern language are framework developers that seek for solutions to provide higher level reuse interfaces, enabling framework-based applications to be built more easily.

Section 2 introduces a simple framework that is used as a running example in the description of the patterns, and several scenarios where specialization aspects can be beneficial. Section 3 presents an overview of the pattern language. Section 4 presents an AspectJ idiom that is used on the implementation of the patterns. Sections 5-10 present the patterns that constitute the pattern language. Section 11 revisits the example framework taking into account the new reuse interface that resulted from all the examples given throughout the patterns. Finally, Section 12 concludes the paper.

2 Example Framework

This section introduces a simple example of a framework, which can be used to build GUI *applications*. An GUI application has *actions* that can be triggered by the UI elements. The action can be either application-specific or provided by the framework. An application may have *menus*, which may contain submenus. The menus may contain either items that trigger application actions or other menus (i.e. the submenus). Implementation-wise there is no distinction between a menu and a submenu (i.e. they are represented by the same class). Figure 3 contains an UML class diagram depicting the classes of the framework's reuse interface (in gray), and an example application (in white) based on the given reuse interface. Below we present Java code that implements the example framework-based application.

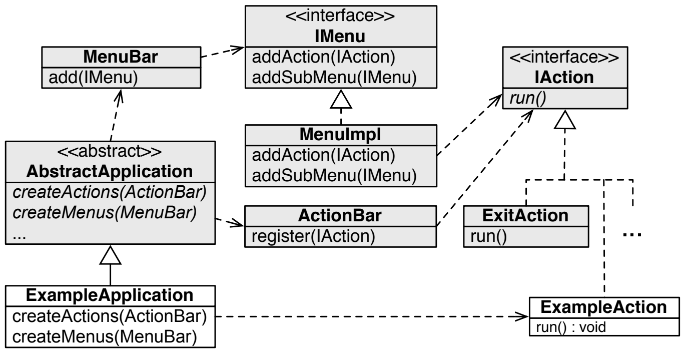


Fig. 3. Reuse interface of the example framework (gray), example application (white).

```
public class ExampleApplication extends AbstractApplication {
  private IAction myaction;
  private IAction exitaction;

  protected void createActions(ActionBar a_bar) {
    myaction = new ExampleAction();
    a_bar.register(myaction);
    exitaction = new ExitAction();
    a_bar.register(exitaction);
  }


  protected void createMenus(MenuBar m_bar) {
    IMenu menu1 = new MenuImpl("Menu1");
    menu1.addAction(exitaction);
    IMenu menu2 = new MenuImpl("Menu2");
    menu2.addAction(myaction);
    menu1.addSubMenu(menu2);
    m_bar.add(menu1);
  }
}
```

```
public class ExampleAction implements IAction {
  void run() {
    // do something
  }
}
```

The main class is a subclass of AbstractApplication. Application developers must be aware that createActions() is executed before createMenus(). The sample framework-based application has two actions, an application-specific one, ExampleAction, and the framework-provided ExitAction. It has a "Menu1", which has the exit action and a submenu "Menu2" that has the application-specific action.

Usage scenarios

The following list presents a set of scenarios where application developers (i.e. the ones who use the framework) may be faced with difficulties. Each of these scenarios is associated with a goal of application developers.

– *Scenario 1, plugging menus.* The application concept *menu* is represented directly by the interface IMenu, which MenuImpl implements. Therefore, it should be easy for an application developer to locate it. However, once the interface/class is known, it is necessary to find out how to plug the menu in the application. Given that the *application* concept is represented abstractly by the AbstractApplication class, one would go to inspect that class, and then, to realize that there is a hook method for the intended purpose (i.e. createMenus(...)). Plugging the menu involves modifying the method body, which may have existing statements. Therefore, in order to implement the goal of plugging a menu, one has to "interfere" with statements pertaining to other goals (i.e. other menus and their contents).

– *Scenario 2, menu context.* The application concept *menu* can be used in two different contexts, either as an *application* menu or as *submenu* of another menu. As explained in Scenario 1, by knowing IMenu and MenuImpl one does not know where and how the menus can be plugged. If one has an existing application menu *m1* and wants that menu to become a submenu of another menu *m2*, besides understanding the subclass of AbstractApplication (Scenario 1) to remove the statement that plugs the menu, there is need to locate where *m2* is instantiated and to know its interface to add *m1* to it. Therefore, changing the context of an existing menu requires changes both in statements pertaining to the original context and in statements pertaining to the new context.

– *Scenario 3, associating actions.* The application concept *action* is represented by the interface IAction. Actions may be associated to *menus*. Suppose that there is an existing application with an action a and a menu m, and that one wants to associate a to m. In order to do so, one has to inspect the hook method createActions() of the subclass of AbstractApplication to find out the instance of a, and then, to modify the hook method createMenus() by finding the instance of m and adding a statement that associates a to it. Therefore, an association between two application elements involves two parts of a module (the subclass of AbstractApplication) which is not directly related to those elements.

Each of the given scenarios can be improved by applying the MODULAR HOT SPOTS pattern language. When addressing a scenario by applying a pattern, it might happen that a scenario with a new problem arises. In these cases, there are other patterns for overcoming the new problems.

3 Pattern Language Overview

Figure 4 gives an overview of MODULAR HOT SPOTS. The diagram contains related patterns and idioms represented in white, while the actual patterns/idioms of the language are represented in gray. Design patterns are represented in ellipses, whereas AspectJ idioms are represented in circles.

Hot spots based on TEMPLATE METHOD [2] are typical starting points for applying the pattern language. It is common that an application framework applies at least one TEMPLATE METHOD on the main class that initializes the application. A TEMPLATE METHOD has one or more hook methods, which have to be overridden by application developers. A COMPOSITION HOOK METHOD (Section 5) is a hook method that exposes an object instantiated by the framework as a parameter, with the purpose of enabling applications to plug objects in the exposed object. While this pattern is not related with the development of MODULAR HOT SPOTS directly, it describes a common solution that hints where it is suitable to have a SELF-PLUGGABLE OBJECT (Section 6). As we will see, COMPOSITION HOOK METHODS are "predictable" and can be completed by a SELF-PLUGGABLE OBJECT, after which the application developer no longer has to deal with those hook methods. In the context of the example framework given in Section 2, this pattern is suitable for improving the *plugging menus* scenario.

A Self-Pluggable Object is a hot spot that enables its adaptations to localize both the creation of an object representing an application element and its composition with another application element. It may be plugged in another Self-Pluggable Object and it may have Composition Hook Methods itself. It can be implemented using a Template Pointcut (Section 4). A Template Pointcut is an AspectJ idiom that combines the idioms Abstract Pointcut and Composite Pointcut [3].

A Multi-Context Self-Pluggable Object (Section 7) is a special kind of Self-Pluggable Object that is suitable in cases when the object can be plugged in different application contexts (elements). The Multi-Context Self-Pluggable Object pattern is suitable for improving the *menu context* scenario. An Abstract Self-Pluggable

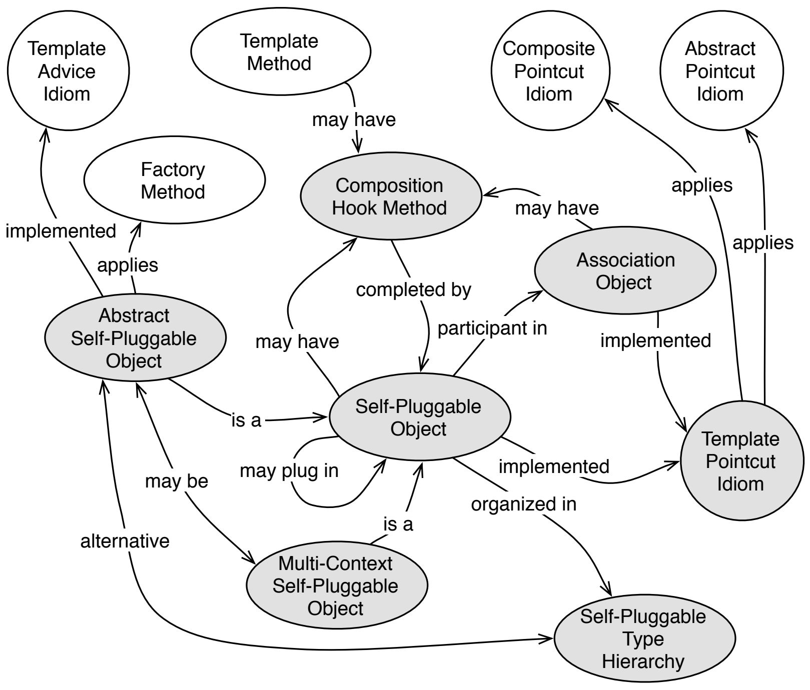


Fig. 4. Modular Hot Spots pattern language (in gray). The elements depicted in white are patterns or idioms previously described by other authors.

Object (Section 8) is a module suitable for structuring a set of related Self-Pluggable Objects, so that the behavior that plugs those objects can be reused. It applies Factory Method [2] and can be implemented using the Template Advice idiom [3]. An alternative to structure a set of related Self-Pluggable Objects is to have a Self-Pluggable Type Hierarchy (Section 9), which merges the implementation of types and the plugging of objects in the applications. The patterns Abstract Self-Pluggable Object and Self-Pluggable Type Hierarchy are two alternatives that are suitable for solving a design problem that can emerge from applying either Self-Pluggable Object or Multi-Context Self-Pluggable Object.

Finally, an Association Object (Section 10) enables to establish associations between Self-Pluggable Objects. This pattern is suitable for improving the *associating actions* scenario.

The examples of applying the patterns are given in Java, using AspectJ as the AOP language. Although the patterns were only experienced in AspectJ, they are not necessarily specific to it. An AOP language for a base object-oriented language, that features method execution pointcuts, abstract aspects, and abstract pointcuts, should be suitable for implementing the patterns. For instance, the patterns should be applicable to AspectC++ [10], AspectJ counterpart for C++.

In the figures that illustrate the solutions, the framework modules are always represented in gray, whereas the white classes represent application modules. Aspects are depicted with a class with stereotype ≪aspect≫. Pointcuts and advices are represented in the method's placeholder using the stereotypes ≪pointcut≫ and ≪advice≫, accordingly. Stereotyped dependencies represent pointcut definitions, where the stereotype name represents the pointcut name.

4 Template Pointcut: an AspectJ Idiom

This section presents an AspectJ idiom referred to as Template Pointcut. Its name results from an analogy with the Template Method pattern. In a Template Method we have a partially implemented method which uses abstract methods that are given by subclasses. In the case of a Template Pointcut, we have a partially defined pointcut within an aspect module that uses abstract pointcuts that are given by the subaspects.

Context

An aspect module transforms (by *weaving*) other modules, which can be either classes or other aspect modules. A reusable abstract aspect is a module from which other aspects inherit (the subaspects), reusing its implementation. The scope of applicability of a reusable aspect may be restricted to a certain kind of base modules. The advantage of doing so is that the reusable aspect may assume certain characteristics of the modules which are going to be transformed. For instance, the reusable aspect may be applicable to all subclasses of a certain class, an therefore the common inherited methods may be safely used by the aspect.

Problem

How to implement a reusable abstract aspect so that its advice can only take effect in a partially defined set of join points, while being able to generalize the commonalities between those join points?

Forces

- The information factored out to the reusable aspect should be maximized.
- The more "black-box" the reusable aspect is, the better.
- The simpler the pointcut definitions in the subaspects are, the better.
- The less one needs to know about the modules that an aspect transforms, the better.

Solution

Implement an abstract aspect containing a Composite Pointcut (the *template*) that is defined as the intersection of certain join points with another Abstract Pointcut (the *hook*). The advice takes effect on the Template Pointcut (Figure 5). Subaspects of the abstract aspect have to define the hook pointcut.

Example

Consider a reusable aspect that can be used to transform classes that inherit from the following abstract class.

```
public abstract class AbstractClass {
   /* ... */
   public String method();
}
```

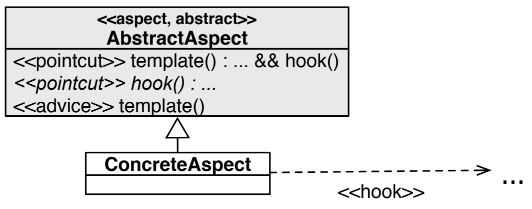


Fig. 5. TEMPLATE POINTCUT idiom.

The following is a reusable aspect with a TEMPLATE POINTCUT defined as the intersection between the execution of method() within subclasses of AbstractAspect and a hook class, which is to be given by the abstract pointcut hook(). The definition of hook() is intended to match a particular subclass of AbstractAspect, which the aspect will transform.

```
public abstract aspect AbstractAspect {
   private pointcut template() :
      within(AbstractClass+) && hook() && execution(String method());

   protected abstract pointcut hook();

   after() returning(String s) : template() {
      /* do something, e.g. */
      System.out.println(s);
   }
}
```

Although the pointcut template() is declared separately, it could be incorporated directly in the advice. Assuming the existence of a subclass of AbstractClass named SomeClass, the code below shows how the aspect could be used for activating the transformation of SomeClass.

```
public aspect ConcreteAspect extends AbstractAspect {
   protected pointcut hook() : target(SomeClass);
}
```

A TEMPLATE POINTCUT is particularly useful to relieve the one who reuses the aspect from understanding details of the modules where the aspect takes effect.

5 Composition Hook Method

Context

TEMPLATE METHOD is an elementary and common pattern for enabling framework specialization, where adaptation is achieved by subclassing.

The role of the hook methods that have to be overridden is often to plug objects on the application.

Problem

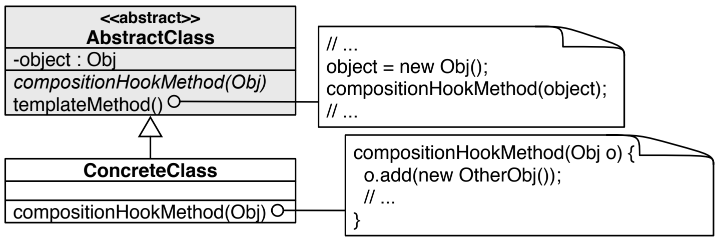
How to define hook methods for the purpose of enabling object plugging, so that they are intuitive to use?

Forces

– Reuse interfaces should be as simple as possible. By reading a hook method signature, it should be intuitive what the method has to do and how.
– The less framework methods that one has to know for building an application, the better.

Solution

Define hook methods that expose in their parameters objects that are instantiated by the framework. These exposed objects are accessed by applications for composing other objects. The intent of a Composition Hook Method (Figure 6) is intuitively given by the method signature, while the way how to plug objects on the exposed object is given by its interface.



```
// ...
object = new Obj();
compositionHookMethod(object);
// ...
```

```
compositionHookMethod(Obj o) {
  o.add(new OtherObj());
  // ...
}
```

Fig. 6. Composition Hook Method pattern.

Example

Considering the example framework, the abstract class AbstractApplication could be something like shown below. The class constructor is the template method, and there are two Composition Hook Methods for plugging actions and menus.

```
public abstract class AbstractApplication {
  private ActionBar a_bar;
  private MenuBar m_bar;

  protected AbstractApplication() {
    a_bar = new ActionBar();
    createActions(a_bar);
    m_bar = new MenuBar();
    createMenus(m_bar);
  }

  protected abstract void createActions(ActionBar a_bar);
  protected abstract void createMenus(MenuBar m_bar);

  /* ... */
}
```

Resulting Context

- There is no need for additional methods whose purpose is to do the object compositions, which are done through the interface of the exposed objects.
- The way how to use hook methods is intuitive by reading their signature.

Known Uses

The main class of a JHotDraw-based application has to override Composition Hook Methods for plugging *menus* and the *tools* that create the figures. A *viewpart* of an application based on Eclipse RCP has a Composition Hook Method for plugging GUI elements.

Related Patterns

The reader may indeed find a Template Method and this pattern very alike. However, the purpose of a Template Method is more generic, and the hook methods may have purposes other than enabling object composition.

By overriding a Composition Hook Method the variation is achieved through the exposed object. The type of objects that can be composed in the exposed objects is known, and there are methods in the objects' interface specifically for that purpose. Therefore, the body of an overridden Composition Hook Method is predictable in what respects to the method invocations on the exposed object. For instance, the only purpose of the exposed object of type ActionBar in the given example is to perform register() calls with objects of type Action as arguments. All the implementations of this Composition Hook Method will be similar across framework-based applications.

A Composition Hook Method may become hidden from application developers, so that they will not need to deal with it when building an application. In order to do so, a Self-Pluggable Object is capable of dismissing the need of overriding the Composition Hook Method.

6 Self-Pluggable Object

Context

Framework classes have Composition Hook Methods.

Problem

How to hide a Composition Hook Method from the reuse interface, so that there is one less framework element that application developers have to know about?

Forces

– Usually the type of the objects we want to plug is easy to find. Finding the way how to plug the objects is the most difficult part, since the framework user has to understand the interface of the object where plugging takes place.
– The less hook methods that have to be known and dealt by application developers, the better.
– The lack of documentation may cause application developers to plug objects in wrong locations, resulting in incorrect uses of the framework.

Solution

Develop an abstract aspect that encapsulates the behavior that creates and plugs the object in a Composition Hook Method. Use a Template Pointcut where the fixed part defines the Composition Hook Method and the variable part (hook) is intended to match a subclass of the template class that owns that method. Application developers use a Self-Pluggable Object (Figure 7) by extending the aspect and defining the hook pointcut on the desired context.

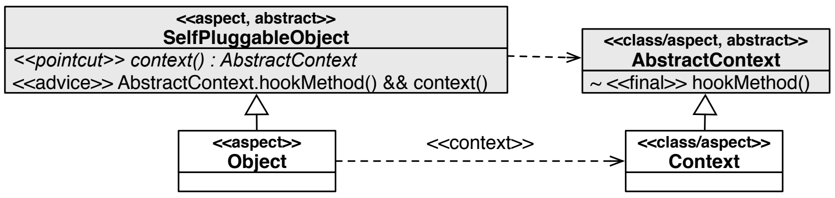


Fig. 7. SELF-PLUGGABLE OBJECT pattern.

Example

This pattern is illustrated by presenting a solution for improving the *plugging menus* scenario given in Section 2. The COMPOSITION HOOK METHOD is AbstractApplication.createMenus(). Since this method will not need to be overridden by applications, it may have reduced visibility and be locked for overriding, as shown below.

```
public abstract class AbstractApplication {
  /* ... */
  final void createMenus(MenuBar m_bar) {   }
}
```

The following is a reusable aspect for plugging menus. The menu name is given in the constructor, while the menu is created by createMenu() using that name. The hook pointcut is application(). The advice creates the menu and plugs it on the MenuBar object parameter of createMenus(..). Using an independent method for creating the menu facilitates the collaboration with other aspects.

```
public abstract aspect Menu {
   private String name;

   public Menu(String name) {
      this.name = name;
   }

   protected abstract pointcut application();

   after(MenuBar mb) :
      within(AbstractApplication+) && application() &&
      execution(void createMenus(MenuBar)) && args(mb) {
         mb.add(createMenu());
      }

   IMenu createMenu() {
      return new MenuImpl(name);
   }
}
```

The main class of an application (subclass of AbstractApplication) does not need to override createMenus(..). ExampleApplication would be given like shown below.

```
public class ExampleApplication extends AbstractApplication {

}
```

In order to plug a menu, a subaspect of Menu has to be defined. The following is an example of how to plug a menu ("Menu1") in ExampleApplication.

```
public aspect Menu1 extends Menu {
   public Menu1() {
      super("Menu1");
   }

   protected pointcut application() : target(ExampleApplication);
}
```

In case the order of multiple Self-Pluggable Objects of the same type is relevant, precedences have to be used to explicitly declare the order in which the several objects are plugged. The following example shows how it could be declared that Menu1 is to be plugged before MenuX.

```
public aspect MenuOrder {
   declare precedence: MenuX, Menu1;
}
```

The precedence declaration may be given in an independent module as shown, but it can also be given together with the other modules.

Resulting Context

- Application developers no longer have to deal with the Composition Hook Method. Instead, they implement an independent aspect, which defines the hook pointcut.
- Objects can be plugged in other objects incrementally, without the need of modify, inspect, or understand, code related to the object where composition takes place.
- Changing the context where the object is composed can be done only by changing the hook pointcut definition.
- Framework-based applications are adaptable without the need of understanding or modifying source code. Removing a Self-Pluggable Object can simply be done by recompiling the application without its module (e.g. deactivating Menu1 as a compilation unit).

Known Uses

Self-Pluggable Objects in JHotDraw can plug *menus*, *tools*, and *undo* on tools. Self-Pluggable Objects in Eclipse RCP can plug the *toolbar*, *perspectives*, and *viewparts* (an application can have several viewparts, which are organized in different perspectives).

Related Patterns

The Decorator pattern [2] is related to Self-Pluggable Object, in the sense that also allows to add behavior to a class modularly. A Decorator adds behavior dynamically to an object by wrapping it. This implies that when adding a Decorator one has to modify the module that instantiates the wrapped object. A Self-Pluggable Object does not require modifying nor inspecting the module where behavior will be added.

A Self-Pluggable Object can be plugged in another Self-Pluggable Object. This can be done by intercepting the creation of objects in order to plug other objects in them, or by completing Composition Hook Methods, which Self-Pluggable Objects may have. A Self-Pluggable Object may be a Multi-Context Self-Pluggable Object if the object can be plugged in different application contexts. A Self-Pluggable Object may be based on an Abstract Self-Pluggable Object if there are multiple subtypes of the pluggable object. A Self-Pluggable Type Hierarchy merges the type implementations with their abstract composition (i.e. plugging). An Association Object enables the establishment of associations between Self-Pluggable Objects.

7 Multi-Context Self-Pluggable Object

Context

A Self-Pluggable Object is an object that plugs itself in a certain application context. However, there are objects which can be plugged in different application contexts.

Problem

How to develop a Self-Pluggable Object that can be plugged in more than one application context?

Forces

- It is appealing to have everything what is possible to do with an object represented in a single module. By knowing about that module, an application developer knows all that can be done with the object.
- If we would have a Self-Pluggable Object for each application context, there would be multiple modules for addressing a single concept.

Solution

Develop an aspect similar to a Self-Pluggable Object, which has one advice for each Composition Hook Method related with an application context. The hook pointcut is used in the different advices. When using the Multi-Context Self-Pluggable Object (Figure 8), the context is given by the module that is matched by the hook pointcut, implying that only the advice related to that application context will take effect.

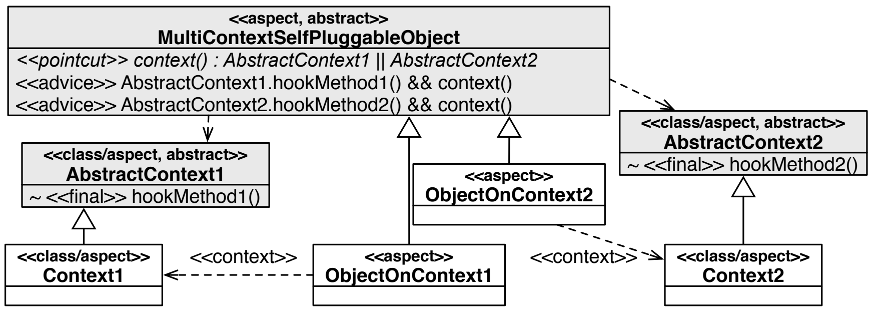

<<aspect, abstract>>
MultiContextSelfPluggableObject
<<pointcut>> context() : AbstractContext1 || AbstractContext2
<<advice>> AbstractContext1.hookMethod1() && context()
<<advice>> AbstractContext2.hookMethod2() && context()
<<class/aspect, abstract>>
AbstractContext1
~ <<final>> hookMethod1()
<<class/aspect, abstract>>
AbstractContext2
~ <<final>> hookMethod2()
<<aspect>>
ObjectOnContext2
<<class/aspect>>
Context1
<<context>>
<<aspect>>
ObjectOnContext1
<<context>>
<<class/aspect>>
Context2

Fig. 8. Multi-Context Self-Pluggable Object pattern.

Example

This pattern is illustrated by evolving the previous example, while addressing the improvement of the *menu context* scenario given in Section 2. Besides the application, a (sub-)menu can also be composed in another menu. Therefore, a menu can be used in more than one application context. The following is a new version of Menu, containing two advices. The first is like in the previous example, while the second is for addressing the composition of menus and sub-menus.

```
public abstract aspect Menu {

   /* to match an extension of either AbstractApplication or Menu */
   protected pointcut context();

   after(MenuBar mb) :
      within(AbstractApplication+) && context() &&
      execution(void createMenus(MenuBar)) && args(mb) {
         mb.add(createMenu());
      }

   after() returning(IMenu m) :
      within(Menu+) && context() &&
      execution(IMenu createMenu()) {
         m.addSubMenu(createMenu());
      }
   /* ... */
}
```

The following module would plug the menu "Menu1" in ExampleApplication (very similar to the example given previously).

```
public aspect Menu1 extends Menu {
   public Menu1() {
      super("Menu1");
   }

   protected pointcut context() : target(ExampleApplication);
}
```

The following module would plug the menu "Menu2" in the "Menu1".

```
public aspect Menu2 extends Menu {
   public Menu2() {
      super("Menu2");
   }

   protected pointcut context() : target(Menu1);
}
```

Resulting Context

- Everything that can be done in an application with an object is achieved through the same module.
- Changing the context where the object is composed, including different context types, can be done just by changing the hook pointcut definition in the object's module.

Known Uses

A MULTI-CONTEXT SELF-PLUGGABLE OBJECT in Eclipse RCP can plug *menus*, whose context may be (a) the application menu bar (conventional menu), (b) a certain *viewpart* (only shown in there), and (c) a certain *viewer* (appears as a pop-up menu).

Related Patterns

A Multi-Context Self-Pluggable Object is a special kind of Self-Pluggable Object.

8 Abstract Self-Pluggable Object

Context

Objects are plugged using Self-Pluggable Objects. A common case in frameworks is that objects of a certain type (e.g. represented by an interface) may be plugged in an application, and therefore, several subtypes of that type can be plugged in the same way.

Problem

When having a hierarchy of types whose objects can be plugged in an application, if we would have a Self-Pluggable Object for each one, there would exist duplicated code, given that all the objects are plugged in the same way. How to generalize the common behavior that is necessary to plug objects of a certain type?

Forces

- Code reuse should be maximized.
- A Self-Pluggable Type Hierarchy is also suitable to structure Self-Pluggable Objects, but this option is not always viable.

Solution

Develop an aspect similar to a Self-Pluggable Object, but with a Template Advice where the creation of the object to be plugged is done by a Factory Method (abstract method). This Abstract Self-Pluggable Object (Figure 9) should not be visible to applications. Develop one abstract aspect inheriting from it for each type to be plugged, where the implementation of the Factory Method returns the proper object. If application-specific objects of that type are allowed to be plugged, develop also an abstract aspect that implements the type but which does not implement the methods of the type, so that they can be given in application modules. An application developer may use one of the visible aspects by extending it and defining the hook pointcut. In case the application-specific type is intended to be implemented, the application developer extends the aspect for that purpose, and in addition to the hook pointcut definition, the type's methods have to be implemented.

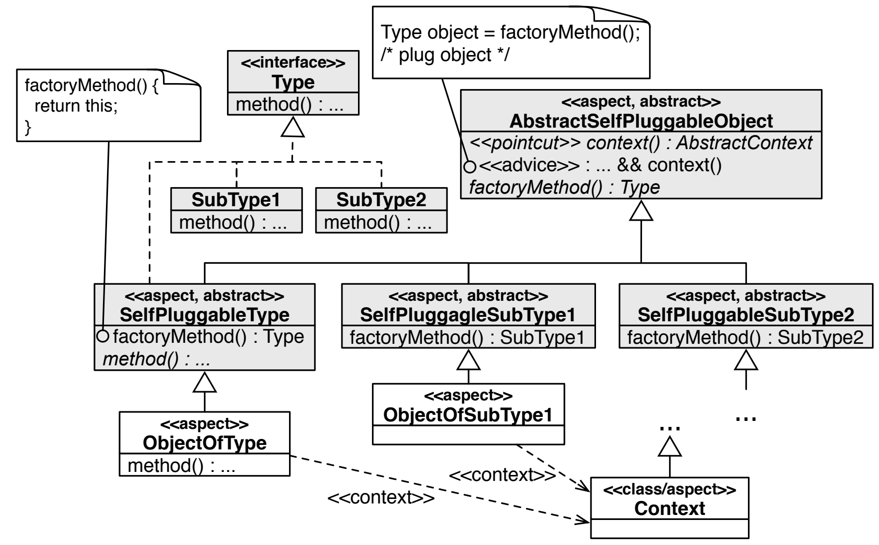


Fig. 9. Abstract Self-Pluggable Object pattern.

Example

This pattern is illustrated with the plugging of *actions* in the example framework. The case of actions is very similar to the *plugging menus* scenario described in Section 2. Applications may include actions by plugging objects of type IAction. The following is an Abstract Self-Pluggable Object for this purpose. Except for the Template Advice, the solution is analogous to a Self-Pluggable Object.

```
abstract aspect AbstractAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
        ab.register(createAction());
      }

   protected abstract IAction createAction();
}
```

The above aspect can be extended by SELF-PLUGGABLE OBJECTS, which can be used by application developers. The following is an example SELF-PLUGGABLE OBJECT, based on the ABSTRACT SELF-PLUGGABLE OBJECT, for addressing the *exit action*. The aspect overrides createAction() for returning an instance of the framework class ExitAction.

```
public abstract aspect Exit extends AbstractAction {
   protected IAction createAction() {
      return new ExitAction();
   }
}
```

The following module illustrates how Exit could be used, by plugging the exit action in ExampleApplication.

```
public aspect ExitOnExample extends Exit {
   protected pointcut application() : target(ExampleApplication);
}
```

The aspect that allows the plugging of application-specific actions could be implemented like shown below. The aspect implements IAction, but it is up to applications to implement the interface methods (run() in this case).

```
public abstract aspect Action extends AbstractAction
   implements IAction {

   public abstract run();

   protected IAction createAction() {
      return this;
   }
}
```

The following module illustrates how Action could be used. In addition to the hook pointcut definition, the method implementation is given.

```
public aspect ExampleAction extends Action {
   public void run() {
      /* application-specific action implementation */
   }

   protected pointcut application() : target(ExampleApplication);
}
```

Resulting Context

– The plugging of objects of a certain type is generalized. Support for new types can be added simply by developing an aspect module that overrides the Factory Method (e.g. as in Exit).
– The code of the modules that extend the Abstract Self-Pluggable Object still has some redundancy given that the implementation of the Factory Methods across the several modules is very similar (only the name of the class changes).
– The number of subaspects grows along with the number of framework-provided type implementations, implying one more framework module for each one (i.e. the aspect).

Known Uses

In Eclipse RCP, an Abstract Self-Pluggable Object can generalize the plugging of *actions*, which may be either chosen from a set of framework-provided actions or implemented by applications. In JHotDraw there is an analogous case for plugging *commands*.

Related Patterns

An Abstract Self-Pluggable Object serves the purpose of structuring a set of related Self-Pluggable Objects, and consists of an alternative to a Self-Pluggable Type Hierarchy.

9 Self-Pluggable Type Hierarchy

Context

An Abstract Self-Pluggable Object is capable of generalizing the plugging of objects of a certain type, implying that there will exist an aspect for each type. All these subaspects are similar and only differ in the object instance returned by the Factory Method.

Problem

How to avoid the existence of all the similar subaspects, and therefore, to reduce the number of framework modules?

Forces

– In solutions based on an Abstract Self-Pluggable Object, the number of subaspects grows along with the number of framework-provided type implementations. The disadvantage is that the solution implies one Self-Pluggable Object for each pluggable type.

Solution

Merge the implementation of a type hierarchy of default components with the Self-Pluggable Objects that implement the composition of objects of that type. In order to do so, develop an aspect similar to a Self-Pluggable Object that is of the top-most type of the hierarchy (i.e. it declares that it implements that type), while it does not implement the type's methods. Develop a subaspect for each subtype, where the type methods are implemented. These aspects may represent partial type implementations by implementing a subset of the type methods, while leaving the remaining methods to applications. Application developers can use the appropriate member of the Self-Pluggable Type Hierarchy (Figure 10) that implements the type they wish to use. If an application has to include its own implementation of the type, it extends the top-most aspect.

Example

This pattern is illustrated with the same case of the *actions* in the example framework, as it consists of an alternative solution to the one given in Section 8.

The following is a new version of Action that can be used in the same way by application developers (exemplified in Section 8). The aspect is of type IAction, and registers itself as an action.

```
public abstract aspect Action implements IAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
         ab.register(this);
      }

   public abstract void run();
}
```

The following is a new version of Exit that can be used in the same way by application developers (as given in Section 8).

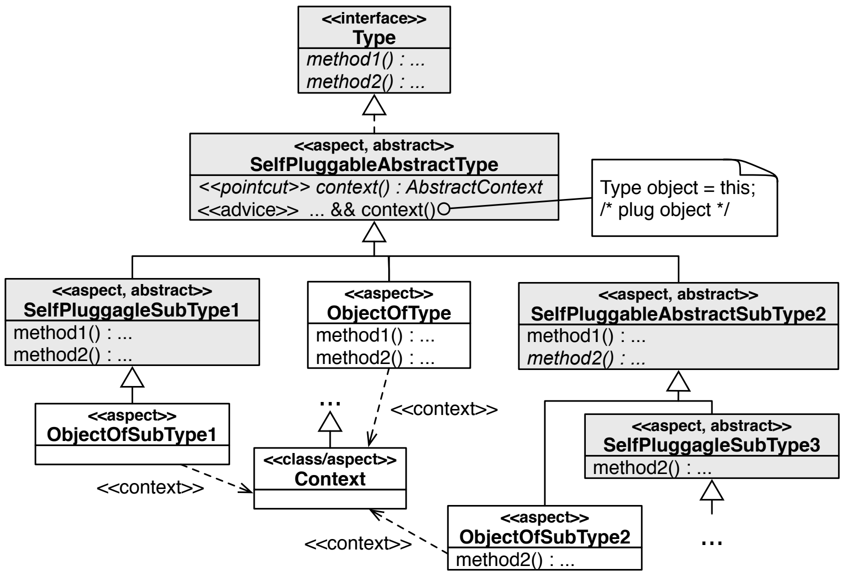


Fig. 10. Self-Pluggable Type Hierarchy pattern.

```
public abstract aspect Exit extends Action {
  public void run() {
    /* the exit action implementation */
  }
}
```

Resulting Context

– Less framework modules when comparing with a solution based on an Abstract Self-Pluggable Object, and more elegant given the nonexistence of all the similar subaspects for each pluggable type.
– The types addressed by the Self-Pluggable Type Hierarchy cannot be instantiated independently.

Known Uses

In JHotDraw, a Self-Pluggable Type Hierarchy is capable of organizing the framework-provided *figures* and *connection figures*. In order to use an application-specific figure, application developers may extend a member of the hierarchy, which may be the top element for implementing a completely new figure, or a lower one in case if the figure is intended to be based on an existing one.

Related Patterns

If merging the implementation of the types with Self-Pluggable Objects is not possible due to some constraint, a solution based on an Abstract Self-Pluggable Object can be used instead.

10 Association Object

Context

Objects are plugged in an application using Self-Pluggable Objects or Multi-Context Self-Pluggable Objects. The objects plugged in an application may need to have other associations between them.

Problem

The plugged objects are not visible to application developers, so that they can define the associations. How to establish an association between two Self-Pluggable Objects?

Forces

– Having the possibility of managing object associations independently is advantageous because application features relying on associations may be plugged and unplugged without modifying other modules.
– Given that the objects are handled by Self-Pluggable Objects, it makes sense to define associations in terms of these modules.

Solution

Develop an abstract aspect, which when made concrete encapsulates an association between two objects — an Association Object (Figure 11). Use two Template Pointcuts to capture the creation of the objects, each one with its own advice. One advice captures and stores a reference to one of the objects, while the other advice uses that reference to establish the association with its captured object. Application developers can establish an association by defining the hook pointcuts on the modules representing the two objects to be associated. An Association

Object can be defined in terms of an Abstract Self-Pluggable Object or the top-level aspect of a Self-Pluggable Type Hierarchy. This enables that the association can be established between any object whose type is a subtype of the type addressed by either the Abstract Self-Pluggable Object or the Self-Pluggable Type Hierarchy.

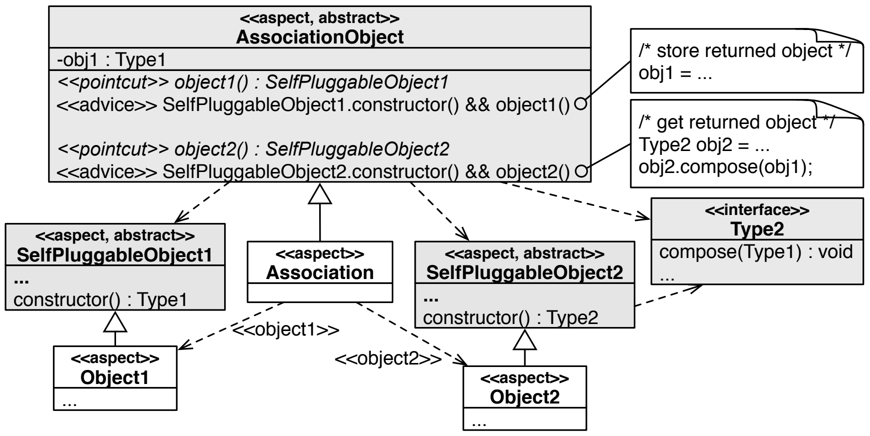


Fig. 11. Association Object pattern.

Example

This pattern is illustrated by presenting a solution for improving the *associating actions* scenario given in Section 2. Below we present the aspect that enables the encapsulation of such associations, assuming the specialization aspect Action from Section 9 and the specialization aspect Menu from Section 7. The first advice captures the instantiation of a subaspect of Action, which is itself an object of type IAction, while the second advice includes the captured action in a menu captured from the execution of createMenu() within a subaspect of Menu.

```
public abstract aspect MenuAction {
   protected abstract pointcut action();
   protected abstract pointcut menu();
   private IAction a;

   after(IAction a) :
      within(Action+) && action() &&
      execution(Action.new(..)) && this(a) {
         this.a = a;
      }

   after() returning(IMenu m) :
      within(Menu+) && menu() &&
      execution(IMenu createMenu()) {
         m.addAction(a);
      }
}
```

Assuming the SELF-PLUGGABLE OBJECTS Menu1 and ExitOnExample given previously, the following aspect implements the association that places the *exit action* on "Menu1".

```
public aspect ExitOnMenu1 extends MenuAction {
   protected pointcut action() : target(ExitOnExample);
   protected pointcut menu() : target(Menu1);
}
```

Resulting Context

- Associations can be encapsulated and managed independently.
- In order to establish an association there is no need to understand the context where the objects are plugged nor any details about their type implementation.

Known Uses

In JHotDraw an ASSOCIATION OBJECT can define the valid source and target *figures* which a *connection figure* may connect. However, the solution is a bit different than the one in the example, since the valid connections are given by overriding a hook method of the connection figure. In Eclipse RCP, ASSOCIATION OBJECTS may link the *actions* and the *toolbar*, the actions and the *menus*, or the *viewparts* and the *perspectives*.

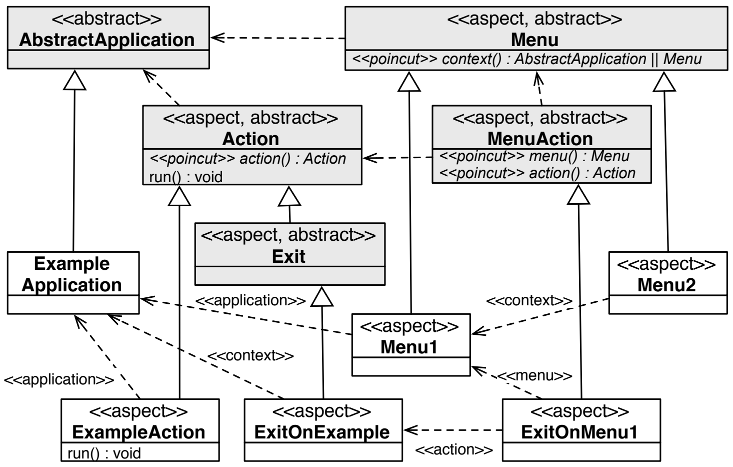
Related Patterns

An ASSOCIATION OBJECT may be adaptable by having COMPOSITION HOOK METHODS, which in turn can be completed by SELF-PLUGGABLE OBJECTS.

11 Example Framework Revisited

This section revisits the example framework, taking into account the MODULAR HOT SPOTS pattern language given throughout sections 6-10.

Figure 12 depicts the new reuse interface after applying the patterns. We can see the several abstract modules (gray) and their abstract pointcuts. Regarding the *menus*, the example of Section 7 is considered (MULTI-CONTEXT SELF-PLUGGABLE OBJECT), instead of the one given in Section 6. Regarding the *actions*, the examples of Section 9 are considered (SELF-PLUGGABLE TYPE HIERARCHY), instead of the ones of Section 8.



<<abstract>> AbstractApplication
<<aspect, abstract>> Menu
<<poincut>> context() : AbstractApplication || Menu
<<aspect, abstract>> Action
<<poincut>> action() : Action
run() : void
<<aspect, abstract>> MenuAction
<<poincut>> menu() : Menu
<<poincut>> action() : Action
<<aspect, abstract>> Exit
Example Application
<<aspect>> Menu2
<<aspect>> Menu1
<<application>>
<<context>>
<<context>>
<<application>>
<<menu>>
<<aspect>> ExampleAction
run() : void
<<aspect>> ExitOnExample
<<aspect>> ExitOnMenu1
<<action>>

Fig. 12. MODULAR HOT SPOTS for the example framework (in gray) and example application (in white).

Figure 12 also depicts the framework-based application based on MODULAR HOT SPOTS that was given throughout the patterns. We can see the several application aspects (inheriting from the specialization aspects) and their pointcut definitions. The following topics briefly compare this solution with the conventional one given in Section 2.

1. Each of the application concepts (i.e. *application*, *menu*, *action*, *exit*, and *menu action*) can be used incrementally, where each concept instance is implemented in an independent module. Throughout the pattern examples a framework-based application was given in the modules ExampleApplication (Section 6), Menu1 and Menu2 (Section 7), ExitOnExample and ExampleAction (Section 8), and ExitOnMenu1 (Section 10).
2. Without source code modification, the application may be compiled with subsets of the modules enumerated in (1), obtaining variants of

the application. This issue is particularly important in the context of *software product-lines*. For instance, one could have a variant of the application without the "Menu2", just by not including that module in the compilation.

3. Application features can be removed without understanding source code, as far as one knows which application elements the modules are representing (a fairly basic information that is easy to maintain). This issue facilitates the *maintenance* and *reengineering* of framework-based applications. For instance, suppose that the application implemented by the modules enumerated in (1) needs to be changed for a new version without the ExampleAction. If the task is given to a programmer that was not the one who developed the application in first place, his or her task becomes facilitated, given that only that module has to be identified and removed, while no understanding of the existing code is necessary.
4. The associations between menus and actions can be independently and non-invasively defined.
5. The two hook methods of AbstractApplication, plus the two methods of Menu, of the conventional reuse interface, no longer have to be dealt with by applications. Instead, there are pointcuts that assume their role. The advantages of the latter is that compositions can take place without modifications and inspection of the target modules.
6. The two classes MenuBar and ActionBar are no longer relevant for the application developer.

The items (1), (2), and (3) are related with the improvement of the *plugging menus* and *menu context* scenarios given in Section 2. Item (4) is related with the improvement of the *associating actions* scenario also given in Section 2.

12 Conclusion

This paper presented Modular Hot Spots, a pattern language for helping on the task of developing framework reuse interfaces with a higher abstraction level concerning the development of framework-based applications. The application of the given patterns relies on aspect-oriented programming primitives. However, the required knowledge of this programming paradigm is small, if we consider the whole set of primitives that these languages offer.

Modular Hot Spots can form a black-box reuse interface with a higher level of abstraction than conventional black-box reuse interfaces.

Black-box frameworks are pointed out as adequate for having an accompanying VISUAL BUILDER [7] for generating framework-based applications from high-level domain-specific descriptions. We argue that such VISUAL BUILDERS can be developed more easily if MODULAR HOT SPOTS are adopted, given that the code of the applications is able to resemble more closely the concepts and relationships of a given domain.

Acknowledgements

We would like to thank our EuroPLOP'08 shepherd Uirá Kulesza, and the Writer's Workshop participants Paul G. Austrem, Dietmar Schütz, Diethelm Bienhaus, and Jürgen Salecker, for the valuable suggestions for improving this paper.

References

1. Eclipse Foundation. AspectJ programming language. http://www.eclipse.org/aspectj, 2007.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
3. S. Hanenberg, A. Schmidmeier, and R. Unland. Aspectj idioms for aspect-oriented software construction. In *8th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2003.
4. R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.
5. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, 1997.
6. J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.
7. D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.
8. A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.
9. SourceForge. JHotDraw framework. http://www.jhotdraw.org, 2006.
10. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceeding of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, 2002.
11. A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 1989.