XV Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)

MODULARIZING FRAMEWORK HOT SPOTS USING ASPECTS

André L. Santos[1] * , Antónia Lopes[1] y Kai Koskimies[2]

1: Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
e-mail: {andre.santos,mal}@di.fc.ul.pt

2: Institute of Software Systems
Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere, Finland
e-mail: kai.koskimies@tut.fi

Palabras clave: aspect oriented software development, frameworks, modularity, reuse

Resumen. *Frameworks are software systems implementing the shared structure and functionality for a family of applications. We propose that the extension points, also known as hot spots, of a framework to be expressed as a set of aspects, creating thus an aspect-oriented wrapper for an object-oriented framework. The benefits of this approach include improved modularity, implying better configurability of application features and stronger reuse of framework code. We illustrate the approach using a popular graphical Java framework, JHotDraw.*

1 INTRODUCTION

The development and use of object-oriented frameworks is a well-known technique of achieving high-levels of reuse in software development. A *framework* is a set of classes that embodies an abstract design for solutions to a family of related problems [1]. It integrates reusable classes and extensible *hot spots*, i.e., design solutions supporting certain variations in the specializations [2]. Hot spots often rely on *design patterns* [3], introducing a point of flexibility in a software system. The process of using a framework for creating an application, known as *framework specialization*, is centered around the identification of the relevant hot spots and their customization.

Typically, an application developer has to use more than one hot spot to implement certain feature or functionality in the application. However, the hot spots are not modularized according to the concerns of the application developers, but scattered around

*On leave at (2) with the support of the Portuguese *Fundação para a Ciência e Tecnologia*

different elements of the framework as dictated by its architecture. This scattering is responsible for one of the main difficulties in using frameworks: it is hard for an application developer to identify the relevant hot spots and understand how they should be specialized.

Aspect-oriented programming (AOP) [4] is a programming paradigm that extends the object-oriented paradigm with mechanisms for modularizing cross-cutting concerns, i.e. concerns that are implemented across more than one implementation module.

In this paper, we propose that the degrees of variability underlying a framework are organized in terms of *specialization concerns*, according to particular goals of specialization, and that such concerns will be realized by *specialization aspects*.

More concretely, our approach consists in the enhancement of frameworks with sets of abstract aspects expressing their hot spots, having in mind a well defined set of *specialization concerns*. The goal is to have modularized implementation of specialization concerns through aspects that extend the specialization aspects, and in this way, minimize the work needed for creating applications. These extensions are modules that can be managed independently, which we refer as *application aspects*.

Our approach is beneficial for framework-based development concerning modularity, configurability, reusability, evolution, and cognitive complexity of framework usage. Furthermore, it is suitable for augmenting existing frameworks with specialization aspects in a non-intrusive way. It preserves the obliviousness principle of AOP [5], since the frameworks are not aware of the specialization aspects.

In this paper, we illustrate our approach using the JHotDraw framework [6] and AspectJ [7], namely by presenting part of AspectJ implementations of two specializations aspects developed for this framework.

The rest of the paper proceeds as follows. Section 2 introduces the JHotDraw case study and the concept of specialization concern. Section 3 describes our approach and portraits how it can be implemented using AspectJ. Section 4 discusses related work, while Section 5 concludes and presents future work.

2 CASE STUDY: JHOTDRAW

JHotDraw is a Java framework for applications that deal with technical and structured graphics. The framework can be considered of type *gray-box*, i.e. its extensibility is based both on *white-box* and *black-box* mechanisms. The first kind relies on class inheritance, while the latter on object composition. JHotDraw allows applications to extend the core functionality with elements such as menus, menu commands, figure types, action listeners, etc. Next, we present a small application based on the examples provided originally with the framework. Using this example, we show the limitations of the traditional approach for specializing frameworks in terms of modularization, and we introduce the concept of *specialization concern*.

2.1 Example of specialization

Figure 1 presents the implementation of a simple application using JHotDraw, consisting of a menu "My Menu" containing two commands: "Hello World" and "Insert Rectangle". The application's root class MyApplication extends the framework class DrawApplication. In order to include application-specific menus, the method createMenus(..) has to be overridden. In line 3, an object representing "My Menu" is instantiated using the framework class CommandMenu. In lines 3-8 and 9-14, the two command objects for "Hello World" and "Insert Rectangle" are instantiated using the framework class AbstractCommand. The first command just opens a dialog message, while the latter adds a new rectangle figure to the current view (drawing window) and refreshes its display. In lines 15-16, the two commands are added to the menu. Finally, in line 17, the menu is added to the menubar (the parameter mb), through the framework method addMenuIfPossible(..). In the bottom part of Figure 1 we can see the resulting application.

```
1  public class MyApplication extends DrawApplication {
2     protected void createMenus(JMenuBar mb) {
3        CommandMenu mymenu = new CommandMenu("My-Menu");
4        AbstractCommand cmd1 = new AbstractCommand("Hello␣World", this) {
5           public void execute() {
6              JOptionPane.showMessageDialog(getRootPane(), "Hi␣there␣world!");
7           }
8        };
9        AbstractCommand cmd2 = new AbstractCommand("Insert␣Rectangle", this) {
10          public void execute() {
11             view().add(new RectangleFigure(new Point(50,50),new Point(100,75)));
12             getRootPane().repaint();
13          }
14       };
15       mymenu.add(cmd1);
16       mymenu.add(cmd2);
17       addMenuIfPossible(mb,mymenu);
18    }
19 }
```

Figure 1: JHotdraw specialization: adding a menu with two commands; the image in the bottom part shows the resulting application

2.2 Specialization concerns

When analyzing software requirements and their implementation in terms of modularization, in current object-oriented systems we are commonly faced with comprehension and evolution difficulties due to the phenomena of *scattering* and *tangling* [8]. That is to say, it is quite common that a single requirement affects multiple design and code mo-

dules (scattering) and it also happens that material pertaining to multiple requirements is interleaved within a single module (tangling). These problems arise due to the incapability of achieving the desired degree of *separation of concerns* [9]. In the context of framework-based development, this is reflected in the fact that, the applications created through specialization do not present the concerns that are relevant for application developers appropriately separated and modularized. We refer to this type of concerns as *specialization concerns*.

For instance, in MyApplication, we can easily identify three specialization concerns: (am) having "My Menu"; (ac_1) having the "Hello World" command under "My Menu"; (ac_2) having the "Insert Rectangle" command under "My Menu". These concerns can be regarded as instances of two more abstract specialization concerns: ($AddMenu$) having a menu in the menu bar; ($AddCommand$) having a command under a given menu.

In MyApplication, the implementation of the three concerns is tangled in a single module (see Figure 2).

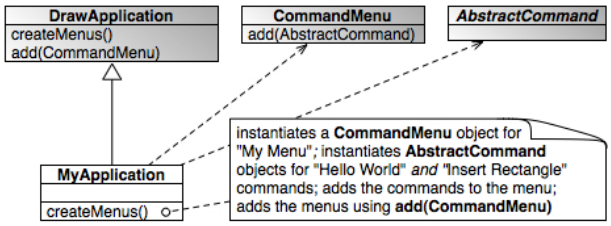

Figure 2: Specializing JHotDraw: adding a menu with two commands (according to the code presented in Figure 1). Framework classes are represented in gray.

One could argue that AbstractCommand objects could be implemented in separate classes, achieving modularization in terms of commands. However, including a command in a specialization would also require that its addition to the intended menu would be coded elsewhere. Therefore, since the inclusion of a command wouldn't be confined to a single module, this would result in code scattering. Nevertheless, there would remain some code tangling, since the calls for adding the two commands to the menu would be interleaved in MyApplication.

Code scattering makes also difficult to localize and modify the code of the different specialization concerns. For instance, in order to remove one of the commands from "My Menu", the application developer has to inspect and modify source code in a module MyApplication where statements associated to several specialization concerns are interleaved. Notice that in the case of a real application, MyApplication.createMenus(..) is likely to have more commands and menus, and that MyApplication has certainly other methods that are relevant for other specialization concerns (e.g. drawing tools).

It is also important to notice that, considering that several specializations make use of menu extensibility, analogous statements to the addToMenuIfPossible(..) and add(..) calls (lines 15-17, Figure 1) are repeatedly coded across different specializations. This hints for

the need of better reuse mechanisms.

A better solution for handling specialization concerns can be achieved using an aspect-oriented approach, which we present in the next section.

3 REPRESENTING SPECIALIZATION CONCERNS AS ASPECTS

This section describes an approach for achieving modularity in the implementation of framework specialization concerns.

Aspect-oriented programming (AOP) [4] introduced the concept of *aspect*, as a unit of modularization in the implementation of software systems. An aspect module can be seen as a *class* with extended capabilities, which is able to add functionality – *advices* – into other identifiable points of a system – *join points*. The join points where an aspect adds functionality are defined in *pointcuts* – predicates that match a set of concrete join points. Analogously to an abstract class, an *abstract aspect* is an aspect that can only be instantiated when extended by a *concrete aspect*. Aspects are intended to be developed in addition to a base (object-oriented) system, and their inclusion is made through a process called *weaving*.

3.1 Augmenting frameworks with aspects

A *specialization aspect* is an abstract aspect that gives support for the implementation of a specialization concern. Specialization aspects are not intended to introduce extra framework functionality, but instead, to provide an improved abstraction for developing specializations, when compared to traditional object-oriented solutions.

Specialization aspects are suitable for *hot-spot-driven development* [10], which consists of an iterative process where hot spots are explicitly identified by domain experts and framework developers, and tested by framework usage for verifying variability requirements. In our approach, each identified hot spot is a candidate for being supported by a specialization aspect.

The key idea of our approach is to enhance existing frameworks with the specialization aspects for giving support to the specialization process and, in this way, minimize the work needed for creating new applications or supporting the evolution of existing ones. A framework specialization involves to develop *application aspects* that extend the relevant specialization aspects according to the requirements (see Figure 3).

The layer of specialization aspects depends on the framework classes. A specialization aspect may depend on other specialization aspects, or it can be a refinement (i.e., an extension) of another specialization aspect. The application aspects for a particular specialization may also have dependencies among them. However, these are intended to be only pointcut dependencies, i.e. an aspect defining a pointcut that matches another aspect or class.

The specialization aspects are intended to be part of the framework, assuming the role of traditional hot spots. However, the possibility of extending the framework using

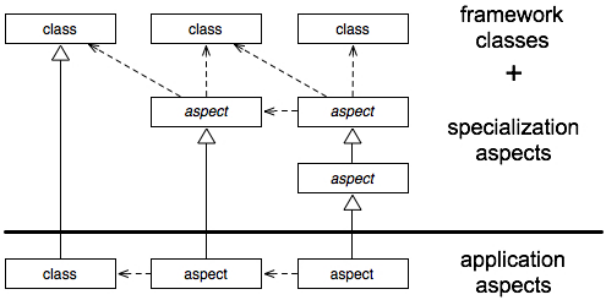

Figure 3: Specialization aspects approach. A class or an aspect in *italics* denotes that it is abstract.

regular object-oriented mechanisms is not excluded, either.

3.2 JHotDraw specialization revisited

Figure 4 illustrates how our approach can be used in the example we have been considering. The solution that we shall present was tested by developing specialization aspects using AspectJ (version 5) on the original JHotDraw framework (version 5.3). The role of each aspect is detailed below, and a possible implementation is given. Some details are omitted, and basic knowledge of AOP mechanisms and AspectJ is required for fully understand the presented code.

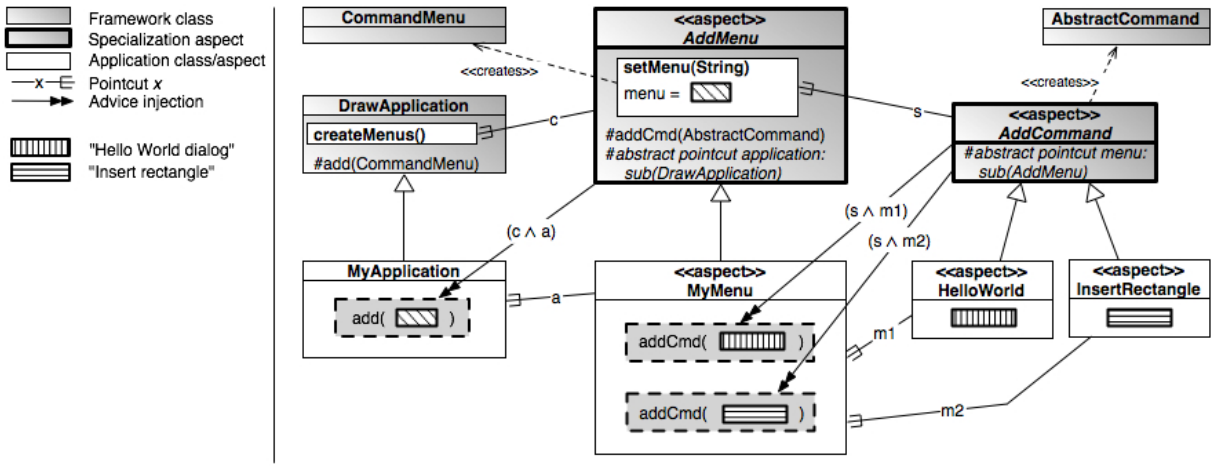

Figure 4: Solution based on specialization aspects for the example given in Section 2.

As discussed in Section 2.2, we have to consider two specialization aspects – AddMenu and AddCommand.

AddMenu supports the inclusion of a menu in an application. Its extensions have to implement the abstract pointcut application (line 2) by matching a subclass of DrawApplication. In lines 7-12, a CommandMenu object is created by setMenu(String) and inserted after the execution of createMenus() using add(CommandMenu).

```
1  public abstract aspect AddMenu {
2     protected abstract pointcut application();  /* match a subclass of DrawApplication */
3     // ...

5     void setMenu(String name) {  menu = new CommandMenu(name);  }

7     after(JMenuBar mb) : execution(void DrawApplication.createMenus(JMenuBar)) && args(mb) &&
8                                       application() {
9        application = (DrawApplication) thisJoinPoint.getTarget();
10       setMenu(name);
11       application.add(menu);
12    }
13 }
```

AddCommand supports the inclusion of a command under a menu. Its extensions have to implement the abstract pointcut menu (line 2) by matching a subaspect of AddMenu, and to implement the method action() (line 3). In lines 6-13, after the executiuon of AddMenu.setMenu(String), an AbstractCommand object is instantiated with the command's action, and inserted in the menu using addCmd(AbstractCommand).

```
1  public abstract aspect AddCommand {
2     protected abstract pointcut menu(); /* match a subaspect of AddMenu */
3     protected abstract void action(); /* command's action */
4     // ...

6     after() : execution(void AddMenu.setMenu(String)) && menu() {
7        AddMenu addMenuAsp = (AddMenu) thisJoinPoint.getThis();
8        application = addMenuAsp.getApplication();
9        Command cmd = new AbstractCommand(name,application) {
10          public void execute() {  action();  }
11       };
12       addMenuAsp.addCmd(cmd);
13    }
14 }
```

In order to obtain an application equivalent to the one presented in Section 2, we have to start by defining a class MyApplication that extends DrawApplication. Notice that it can specialize other parts of the framework normally.

```
1  public class MyApplication extends DrawApplication {
2     // specialize other parts of the framework
3  }
```

Moreover, we have to develop three application aspects – MyMenu, HelloWorld, InsertRectangle – corresponding to the concerns am, ac_1 and ac_2 presented in Section 2.2.

MyMenu extends AddMenu, parameterizing it with the menu name (line 2) and defining the required pointcut on MyApplication (line 4).

```
1  public aspect MyMenu extends AddMenu {
2     public MyMenu() {  super("My-Menu");  }

4     protected pointcut application() : target(MyApplication);
5  }
```

HelloWorld extends AddCommand, parameterizing it with the command name (line 2), defining the required pointcut on MyMenu (line 4), and implementing the hello world dialog (lines 6-8).

```
1  public aspect HelloWorld extends AddCommand {
2     public HelloWorld() {  super("Hello World!");  }
3
4     protected pointcut menu() : target(MyMenu);
5
6     protected void action() {
7        JOptionPane.showMessageDialog(getApplication(), "Hi there world!");
8     }
9  }
```

InsertRectangle extends AddCommand, defining the required pointcut on MyMenu and implementing the insertion of a rectangle on the view. The implementation would be similar to HelloWorld, just differing in the implementation of action().

3.3 Benefits

Based on the presented solution and recalling the specialization concerns pointed out in Section 2 ("My Menu", "Hello World" and "Insert Rectangle"), our approach brings benefits concerning modularity, configurability, reusability, evolution, and cognitive complexity of framework usage.

Modularity. The implementation of each specialization concern is confined to its own independent module (application aspect), eliminating the existing cross-cutting nature in the object-oriented version. Supposing a new concern "Another Menu", in order for instance to change the "Hello World" command to this new menu, the application developer simply modifies the pointcut menu to match the subaspect of AddMenu that handles "Another Menu". **Configurability.** Application aspects can be composed transparently at weaving time (i.e. compilation time) without the need of source code modifications. Although the pointcuts between application aspects can be considered dependencies, they are weak dependencies. If an aspect is not included in the weaving, the other aspects that have pointcut dependencies to it, simply have no effect concerning those poincuts. For instance, it is possible to remove the "Hello World" command just by not including the HelloWorld aspect in the application weaving.

Reusability. A part of analogous statements within the specializations in the object-oriented version is able to be implemented in the specialization aspects. The specialization aspects factor out the generalizable parts of the implementation of generic specialization concerns, leaving just "pure" application-specific issues to be implemented by the applications. For instance, notice that the calls CommandMenu.add(..) and DrawApplication.addMenuIfPossible(..) are no longer performed by the specializations.

Evolution. Specializations have lower coupling to the framework, when comparing to the object-oriented version. Concerning extensibility, the application aspects are only dependent on the specialization aspects they extend, since the hot spot became modularized.

For instance, in order to add a menu in the object-oriented version, the specialization is dependent on the framework classes DrawApplication and CommandMenu, as well as on the methods DrawApplication.createMenus(..) and DrawApplication.addMenuIfPossible(..). In our solution, the specialization is only dependent on the specialization aspect AddMenu and its pointcut menu. If specializations have less dependencies to the framework, the latter can evolve more easily. For example, the internal framework mechanisms for including the menus could change completely without affecting the specializations. Only the specialization aspects that involve menus would have to be modified.

Cognitive complexity. Based on the above points, framework specializations become more easily developed and managed by framework developers. Due to modularization, a specialization concern is easier to trace, since the name of the module itself reflects the concern. Due to configurability, one can effectively modify an application without understanding its implementation. Due to the pointed out issues on reusability and evolution, the application developer has to know and understand less framework details, in order to be able to specialize it, as well as when adapting specializations to newer versions of the framework.

4 RELATED WORK

The work in [11] compares object-oriented and aspect-oriented implementations of the GoF patterns [3], using the concrete technologies of Java and AspectJ. The different implementations are analyzed and aspect-orientation has advantage in properties like locality, reusability, composition transparency and (un)pluggability. The AJHotDraw [12] consists of a refactoring of the JHotDraw framework to aspects using AspectJ. These approaches are revolutionary, in a sense that they imply framework re-implemention. On the other hand, our approach is evolutionary, since it is suitable for being applied in existing frameworks.

JavaFrames [13] is a tool capable of processing formal descriptions of framework specialization patterns, in order to provide task-based assistance in their instantiation. The tool aids the specialization of the framework by guiding and enforcing constraints on the extension of hot spots. The Strathcona tool [14] is able to recommend framework usage examples, based on an existing source code repository. These approaches address the difficulty of specializing a framework, by providing tool support that assists the process. Our approach, intends to address the difficulty on framework specialization by raising the abstraction level of framework specializations, not requiring special tool support.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented an aspect-oriented approach for framework-based development that is able to modularize applications according to specialization concerns. In addition to modularity, we showed that the approach brings benefits in configurability, reusability, evolution, and cognitive complexity of framework usage.

In order to have a proof of concept, the technique was successfully applied for supporting a part of JHotDraw's extensibility mechanisms using AspectJ. The work on the case study will continue, in order to evaluate to which extent is possible to apply the technique.

REFERENCES

[1] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1988.

[2] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., 1995.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, 1997.

[5] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, 2004.

[6] JHotDraw framework. http://www.jhotdraw.org, 2006.

[7] Eclipse Foundation. AspectJ programming language. http://www.eclipse.org/aspectj, 2006.

[8] Peri Tarr, Harold Ossher, Jr. Stanley M. Sutton, and William Harrison. N degrees of separation: Multi-dimensional separation of concerns. In *Aspect-Oriented Software Development*, chapter 3, pages 37–61. Addison-Wesley, 2004.

[9] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[10] Wolfgang Pree. Hot-spot-driven development. In *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc., 1999.

[11] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2002.

[12] AJHotDraw project. http://ajhotdraw.sourceforge.net/, 2006.

[13] Juha Hautamäki. *Pattern-Based Tool Support for Frameworks: Towards Architecture-Oriented Software Development Environment*. PhD thesis, Tampere University of Technology, 2005.

[14] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005.