

Aspects as Specialization Units for Framework-based SPLs

André L. Santos^{1*} and Kai Koskimies²
{firstname.lastname}@tut.fi

¹Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal

²Institute of Software Systems, Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere, Finland

Abstract

A popular technique to implement software product-lines is through object-oriented application frameworks. A major problem with application frameworks is the unstructured, cross-cutting character of their specialization interfaces, and the weak connections between the specialization goals of the application developer and the different parts of the specialization interface. In this position paper, we propose an approach for supporting the specialization of object-oriented frameworks, consisting in representing the variability and the specializations using aspects. In addition to the framework classes, abstract aspects are provided as framework hot spots. Specialization units are implemented by concrete aspects which extend those hot spots. The proposed method can be used to create a modular specialization interface for an existing object-oriented framework in a non-intrusive way. An illustrative example is presented and the potential advantages of the proposed approach are discussed.

1 Introduction

Software Product-Lines (SPLs) have proven to be adequate for achieving large-scale software reuse and reduced time-to-market [2]. SPL development can be divided into domain engineering and application engineering. The domain engineering activity consists of the development and maintenance of a Product Line Platform (PLP), containing the common assets and variability support for the product family associated with the SPL. On the other hand, application engineering refers to the derivation of specific products based on the PLP. One of the problems of application engineering, as pointed out in [2], is the insufficient modularization of the PLP and its extension interfaces.

Frameworks are a popular approach for implementing SPLs [3]. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems [5]. Framework specialization is the process of using classes of the framework for implementing a

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

specific application. In framework-based SPL development, domain engineering deals with the development of the classes that constitute the framework (PLP), while application engineering deals with the use of those classes for producing framework specializations (specific applications). Hot spots [13] are identifiable extensible parts of a framework which allow the configuration or introduction of application-specific assets in a specialization.

A design pattern is a description of a possible solution for a commonly occurring design problem [6]. Design patterns are an adequate method for documenting extensible parts of a framework [9]. Recently, the concept of a specialization pattern [8] has been introduced as a variant of design patterns, especially intended for describing framework extension points related to high-level specialization goals. In contrast to design patterns, specialization patterns are typically framework-specific.

However, whether design patterns or specialization patterns are used to mark the extension points of a framework, the problems related to the unmodularity of the specialization interface remain. In particular, these problems are related to:

- **Locality** - The resulting code of the pattern instantiation is scattered among several participant classes. This implies that the pattern instantiation is not modular, bringing difficulties in its localization and modification.
- **Traceability** - Also due to code scattering, it is difficult to trace the pattern instantiation, specially when the participant classes' code is modified after the instantiation.
- **Reversibility** - Having patterns with common participant classes, or simply modified source code, makes the unplugging of the pattern instance a non-trivial task.
- **Reusability** - Concrete pattern instantiations contain common code statements which can not be reused due to limitations in programming language's constructs.

Aspect-Oriented Programming (AOP) [10] is a programming paradigm which extends the object-oriented paradigm with primitives for allowing modularization of the so-called cross-cutting concerns - a software concern, typically associated with a requirement, whose implementation is spread across more than one decomposition units (e.g. a class).

This position paper proposes the adoption of aspects as specialization units for frameworks serving as PLPs. In order to give support for application engineering goals, abstract aspects are developed as hot spots, providing the generalizable parts of the implementations related to the goals. The implementation of application-specific issues is achieved via abstract aspect extension. When deriving a specific product from the PLP, a concrete aspect extended from a hot spot constitutes a specialization unit.

We argue that this approach facilitates the solving of the problems discussed above, and bridges domain and application engineering by introducing software units that map application engineering goals to certain parts of the PLP. Our approach is suitable for existing frameworks, which need not be adapted for applying our technique.

The rest of the paper proceeds as follows. In the next section we discuss briefly an existing approach to represent pattern-like structures as aspects. The proposed approach is conceptually described in section 3, while section 4 presents an illustrative example of applying the technique in practice. Related work is presented in section 5, and section 6 concludes and presents future directions.

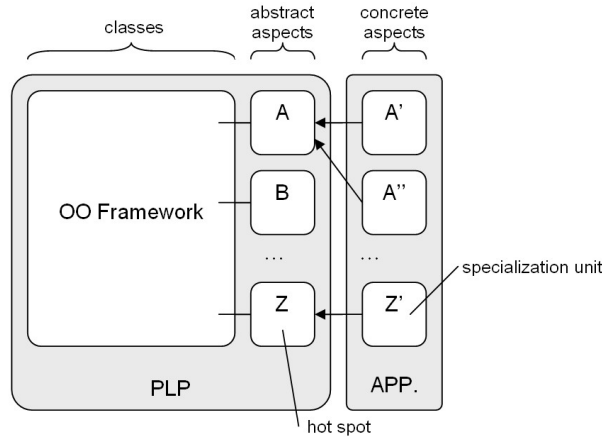


Figure 1: Aspects as specialization units in framework specialization

2 Aspects and design patterns

A design pattern typically involves several participant classes, and therefore its instantiation has a cross-cutting nature in an object-oriented implementation. The work in [7] compares object-oriented and aspect-oriented design pattern implementations, using the concrete technologies of Java and AspectJ [4], accordingly. An analysis of the GoF patterns' [6] implementations is described, concerning the benefits of aspect-orientation in properties like locality, reusability, composition transparency and (un)pluggability. The implementation of the well-known Observer pattern [6] is described in detail, showing evidence on the benefits of the aspect-oriented implementation. A generic part of the Observer mechanism between “subject” and “observer” objects is implemented as an abstract aspect, allowing extensions by concrete aspects to be made, in order to instantiate the pattern in specific situations. The aspect extension only has to define which objects are “subject” or “observer” for a particular case, which actions should consist of subject changes, and which actions to perform when those changes occur.

The advantages of this solution are related to the fact that the pattern instance is confined to a single code module - the concrete aspect extension - without having scattering or code tangling. Modifications on the instantiation of the pattern or its removal become easily managed. These advantages of aspects for design pattern implementation/instantiation motivated the approach proposed in this paper.

3 Specialization units as aspects

Based on the described difficulties related to the use of design patterns for framework specialization, and on the advantages of aspect-orientation in design patterns, we will now describe our approach for framework-based SPL engineering. We assume that the PLP has a set of reasonably well-defined specialization goals associated to it, potentially able of being described by design patterns. Considering domain engineering, in addition to the framework classes, abstract aspects are developed for giving support to the implementation of applica-

tion engineering goals. Analogously to an abstract class, an abstract aspect is only effectively in use when extended by a concrete aspect¹. These abstract aspects represent framework hot spots. When comparing to the use of the framework directly, they will provide a higher level of abstraction for specialization implementation, since their purpose is to implement generalizable parts of a specialization goal. From an application engineering perspective, certain hot spots are going to be selected according to specific specialization goals, and the development of concrete aspects extending the corresponding abstract aspects will constitute the product derivation. Each of the concrete aspects is considered as a specialization unit, which is implemented in a single code module. The approach is conceptually illustrated in figure 1.

This approach intends to overcome the difficulties in managing object-oriented specialization goal implementations, and to provide a higher-level development abstraction for SPL application engineering. The proposed specialization units provide a goal-oriented mechanism for product derivation and posterior maintenance.

4 Example

For the purpose of illustration of the described approach, let's consider a toy SPL whose application domain consists of GUI applications with a menu bar (a Java Swing's `JMenuBar`² object) and a tree displayed graphically. The variability in the product family concerns the menus and menu items that can exist in the application. Derived products can add application-specific menus and menu items which execute application-specific actions.

As a common menu within the product family, the PLP contains a class `JMenuSPL` which extends `JMenuBar`, introducing a *Default* menu (class `JMenu`) containing a menu item *Expand All* (class `JMenuItem`) which expands all nodes in the tree (class `ActionListener`). Roughly, this constitutes the framework of the PLP, and is illustrated in the left slice of figure 2. From an application engineering viewpoint, let's suppose two established specialization goals:

1. Include an application-specific menu
2. Include an application-specific menu item which executes an action

As framework hot spots, two abstract aspects associated with goals 1 (*Menu*) and 2 (*Item*) are included in the PLP. These abstract aspects implement the generalizable parts of the implementation of the goals. The full implementation is achieved through extensions of these aspects, which provide the necessary application-specific information for a particular goal. These hot spots are illustrated in the center slice of figure 2.

Finally, let's consider a simple specialization of our SPL with the following goals:

1. Add a menu *Extra*
2. Add a menu item *Show Message* in the menu *Extra* which shows a message in a pop-up window
3. Add a menu item *Collapse Sports* in the menu *Extra* which collapses the *sports* node of the tree

The implementation of this specialization consists of an aspect `MenuExtra` which extends `Menu` for implementing goal 1, and two aspects, `ItemMessage` and `ItemCollapse`, for implementing goals 2 and 3, accordingly. See illustration on the right slice of figure 2.

¹The semantics of AspectJ is assumed

²See tutorial in http://www.onjava.com/pub/a/onjava/excerpt/swing_14/index1.html

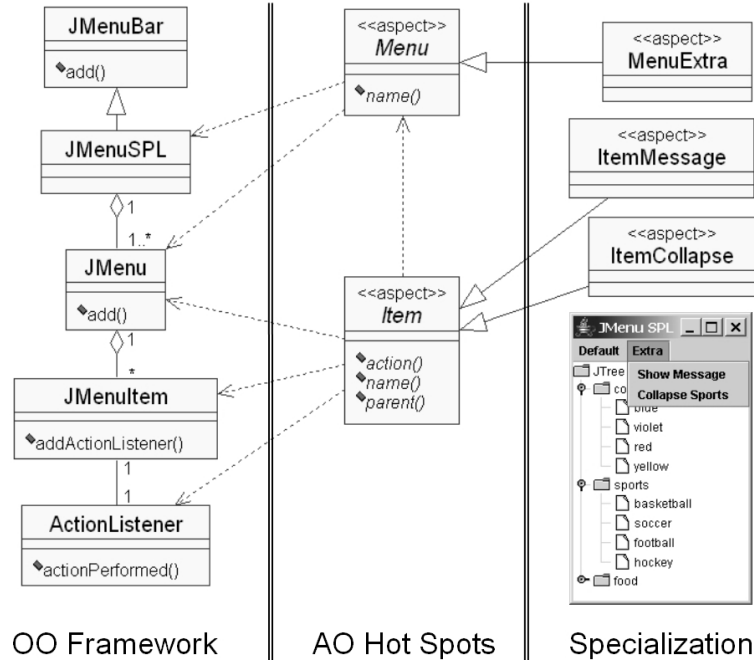


Figure 2: Specialization by extensions of abstract aspects

In order to give an idea of the involved programming abstractions, figure 3 presents AspectJ code for the implementation of the abstract aspect `Item` (lines 1-25), and its extension `ItemMessage` (lines 27-35).

For the following explanation, some knowledge of AspectJ is required. We can see in lines 2-4 the abstract methods of `Item` that have to be implemented, providing the application-specific action for the item, its name, and the name of its parent menu, respectively. Lines 6-10 define the necessary object of type `ActionListener` to associate the action with a menu item, which invokes the `action()` method that will be defined in the aspect's extensions. Lines 12-13 define the `newMenu` pointcut which intersects the creation of `JMenu` objects within the `Menu` aspect (responsible for introducing application-specific menus). Lines 15-24 define an advice associated with `newMenu`, which basically checks if the `JMenu` object that is being created has the name of the item's parent menu (`parentMenu()` method), in order to include the item in it (with name given by `itemName()`), or otherwise proceeds with normal execution. Considering now the `ItemMessage` concrete aspect, it simply defines the action for the item in lines 28-32, its name in line 33, and the name of its parent menu in line 34.

Discussion. Considering the whole sample specialization, notice that each goal was implemented as a single aspect extension, which obviously brings advantages in terms of locality and traceability of the implementation of specialization goals. Reversibility is easily achieved since removing a goal's implementation is done simply by not including the corresponding aspect in the weaving process. Another important issue in terms of the specialization process is that goal implementations are composed transparently, since the aspects that implement the goal are only coupled to the corresponding hot spot. In order to illustrate

```

1 public abstract aspect Item {
2     protected abstract void action();
3     protected abstract String itemName();
4     protected abstract String parentMenu();

6     ActionListener action = new ActionListener() {
7         public void actionPerformed(ActionEvent event) {
8             action();
9         }
10    };

12    pointcut newMenu(String name) :
13        call(JMenu.new(String)) && within(Menu) && args(name);

15    JMenu around(String name) : newMenu(name) {
16        if(((Menu) thisJoinPoint.getThis()).name().equals(parentMenu())) {
17            JMenu menu = proceed(name);
18            JMenuItem item = new JMenuItem(itemName());
19            item.addActionListener(action);
20            menu.add(item);
21            return menu;
22        }
23        return proceed(name);
24    }
25 }

27 public aspect ItemMessage extends Item {
28     protected void action() {
29         JDialog dialog =
30             (new JOptionPane()).createDialog(JMenuSPL.frame,"Moro!");
31         dialog.show();
32     }
33     protected String itemName() { return "Show_Message"; }
34     protected String parentMenu() { return "Extra"; }
35 }

```

Figure 3: Abstract aspect Item and the extension by the concrete ItemMessage aspect

this, imagine for instance the implementation of `ItemCollapse` which is not presented in the paper. Although the implementations of goals 2 and 3 overlap in accessing the same `JMenu` object which resulted from goal 1, this is achieved transparently from the point of view of the product specialization. Finally, there are gains in terms of reusability, since the common issues concerning the specialization goals were able to be implemented within the hot spots.

5 Related work

Design patterns as framework specialization units seem to be adequate for obtaining the implementation of specific goals in SPL product derivation. For example, JavaFrames [12] is a tool capable of processing formal descriptions of design (and specialization) patterns, in order to provide assistance to the developer in their instantiation, based on implementation tasks guidance. However, even though this approach makes the application development easier, many of the problems discussed in section 1 remain.

While the approach proposed in this paper intends to be suitable for existing PLPs (specifically, OO frameworks), there are other existing approaches related with the combination of aspects (or related trends) and SPL development, which imply the development of proper PLPs for applying the technique. For example, the work in [11] proposes the combination of frame technology with aspects in the PLP for achieving product variability, while the feature-oriented programming (FOA) approach [1] is based on having implementation modules of the PLP decomposed by feature, in order to be able to obtain different product configurations. Although our approach relies more on existing framework design, an interesting open question is to what extent abstract aspects, in our sense, could be derived using these techniques.

6 Conclusion and future work

Aspect-orientation is advantageous for design pattern instantiation [7]. Considering pattern instantiation as a mean to implement framework specialization goals, the approach proposed in this paper provides an aspect-oriented strategy for achieving a modularized specialization interface of an existing framework, based on application engineering goals. Having modularized specialization units brings benefits in terms of PLP management and SPL application engineering.

In order to have a proof of concept for the proposed approach, small-scale experiments were made using a small framework as a toy SPL. However, its suitability for real framework-based PLPs has to be evaluated, as well as the adequacy of currently available aspect-oriented constructs for applying them in framework specialization. On the other hand, the proposed technique could potentially work better if the framework where applied was originally designed taking into account aspect-oriented specialization mechanisms.

Considering the approaches for tool-assisted instantiation of design patterns for object-oriented framework specialization (e.g. [12]), the combination of the proposed technique with such development environment seems to be adequate, since they share the same objective in a compatible way - the specialization of object-oriented frameworks.

References

- [1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003.
- [2] J. Bosch. Product-line architectures in industry: a case study. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, 1999.
- [3] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., 2000.
- [4] Eclipse Foundation. AspectJ programming language. <http://www.eclipse.org/aspectj>, 2005.
- [5] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc., 1999.

- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [7] J. Hannemann and G. 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.
- [8] J. Hautamäki. *Pattern-Based Tool Support for Frameworks: Towards Architecture-Oriented Software Development Environment*. PhD thesis, Tampere University of Technology, 2005.
- [9] R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, 1992.
- [10] 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.
- [11] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *Fifth International Workshop on Product Family Engineering*, 2003.
- [12] Practise Research Group. JavaFrames tool. <http://practise.cs.tut.fi/project.php?project=fred>, 2005.
- [13] W. Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., 1995.