Framework Specialization Aspects

André L. Santos *
Institute of Software Systems
Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere
FINLAND
andre.santos@tut.fi

Antónia Lopes
Department of Informatics+
Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa
PORTUGAL
mal@di.fc.ul.pt

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

Abstract

Object-oriented frameworks play an important role in different kinds of software, such as product-lines, middleware, GUI components, IDEs, etc. Over the past recent years, fundamentals of framework design stabilized around the adoption of design patterns. However, major difficulties concerning framework learning and usage are still evident, and constitute a burden for those who have to deal with it. This paper proposes an approach that aims to facilitate framework usage, based on the concept of *specialization aspect*. We show how framework hot-spots can be modularized in terms of specialization aspects, and how these can give support for specializing a framework in a step-wise way. The approach is conservative, in the sense that specialization aspects can be developed for an existing framework "as is". In order to support these claims, a case study has been carried out by applying the technique on the JHotDraw graphical framework.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Object-oriented design methods; D.3.3 [*Language Constructs and Features*]: Frameworks, Patterns

General Terms Design, Languages

Keywords Framework Specialization, Patterns, Hot-Spots, Aspect-Oriented Programming, Step-Wise Refinement

1. INTRODUCTION

An object-oriented *framework* consists of a set of classes that embodies an abstract design for solutions to a family of related problems [22]. Frameworks play an important role in different kinds of software, such as product-lines [5, 6], middleware (e.g. J2EE), GUI components (e.g. MFCs - Microsoft Foundation Classes), Integrated Development Environments (e.g. Eclipse), etc.

Despite the wide adoption of frameworks, there are several difficulties persisting in framework-based development, both in the development of frameworks themselves and in the development of applications based on frameworks (known as *framework specializations*). Two major reasons for the problems encountered in framework-based development are due to the evolution of frameworks and to the complexity of frameworks as reusable software assets.

Frameworks are long-lived entities that have to evolve according to the evolution of their domain requirements [7]. Changes in a framework, which either introduce new features or correct defects, are likely to imply that existing specializations become incompatible and have to be upgraded. Although this issue depends on the number of existing specializations and the nature of the changes, the cost of upgrading all applications tends to be high.

A main challenge in using a framework is to understand how the application-specific code should be given. Learning a framework of reasonable size is considered a difficult and time-consuming task [12]. Frameworks have a steep learning curve [33] and they must have good documentation support in order to be used as intended. However, the cost of documenting frameworks is high [7] and there are no widely adopted standards for doing it. As a result, in industry practice, many frameworks are not accompanied of appropriate and up-to-date documentation [5]. Most of framework documentation is based on *cookbooks* [25], i.e., informal descriptions of framework extension points providing the "recipes" on how to use the framework.

The development of applications based on frameworks takes place at certain extension points called *hot-spots* [35]. These points are adapted in framework specializations, allowing the implementation of application-specific features. An important factor of complexity in framework usage is the scattered and tangled nature of framework hot-spots. On one hand, the implementation of a feature, achieved through the adaptation of an hot-spot, typically involves several classes (scattering [24]). On the other hand, it is common that a single class participates in several hot-spots (tangling [24]), and hence, the implementation of different features may require the adaptation of the same class.

The contribution of this paper is a technique based on aspect-oriented programming (AOP) that supports the development of framework specializations with high *feature cohesion* [27]. The key idea is to have design mechanisms that provide the capability to encapsulate all the necessary code that supports a certain framework feature into a single module. These design mechanisms, defined in terms of abstract aspects, are what we have called *specialization aspects*. Our proposal is to enhance frameworks with specialization aspects expressing their hot-spots. Specializations can then be implemented through the extension of these specialization aspects. In this way, a one-to-one mapping of features to implementation

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

modules is achieved. Additionally, the approach has benefits concerning the cognitive complexity of framework usage, reuse, and evolution of both frameworks and specializations.

Our approach raises the abstraction level of framework specialization in a conservative and domain-independent manner, contrasting with revolutionary or domain-specific language approaches (see Sections 6.3 and 6.4). The approach is conservative in the sense that existing frameworks are suitable for being enhanced with specialization aspects without modifications, and domain-independent, in the sense that it can be applied in a framework of any domain using general-purpose technologies.

We carried out a case study by applying the technique on JHotDraw [38], a popular graphical editor framework generally considered as a good example of applying design patterns. We developed specialization aspects for the main extension points of the framework using AspectJ [23]. An interesting characteristic of these specialization aspects is that in terms of AOP they only rely on exact method pointcuts. On the other hand, the development of specializations only has to deal with exact class pointcuts, in addition to the traditional means, such as extension, realization, method overriding. This means that the necessary AOP primitives for applying our approach are a small subset of AspectJ, which together with the conservative characteristic of the approach, would facilitate its adoption in framework engineering.

The main contributions of this paper are (i) an analysis of the relation between hot-spot design and scattering/tangling, (ii) a technique for modularizing hot-spots, (iii) an evaluation of the approach against traditional object-oriented specialization, and (iv) a case study realizing the approach.

The paper proceeds as follows. Section 2 addresses the problem of scattering and tangling in framework hot-spots. Section 3 presents the concept of specialization aspects. Section 4 evaluates our approach. Section 5 presents a case study on the JHotDraw framework. Section 6 discusses related work, and Section 7 concludes.

2. SCATTERED AND TANGLED HOT-SPOTS

Framework hot-spots typically rely on *design patterns*, such as the ones described in the popular GoF catalog [14]. In general, hot-spots consist of adaptations or compositions of design patterns for a framework-specific context.

The methods of the framework classes that are involved in a hot-spot can be categorized as *hook methods* or *template methods*. Hook methods can be viewed as placeholders for variable parts that are invoked by more complex framework methods – the template methods. Template methods define abstract behavior or generic interactions between objects. In order to use the framework for a certain purpose, a specialization defines subclasses that override the hook methods.

Design patterns can follow either the *unification* or *separation* construction principle [36]. A class that contains one or more template methods is called a *template class*, while a class that contains one or more hook methods is called a *hook class*. Patterns that have classes that are simultaneously template and hook classes are based on unification (e.g. *Template Method* [14]). In this case, adaptation is achieved through inheritance. In contrast, when template and hook classes are separated, the pattern is based on separation (e.g. *Strategy* [14]) and adaptations are achieved through object composition.

In Figure 1, we present an example of the extension of two hot-spots of a framework with the classes Context and Strategy. In this framework, on one hand, the Context class represents an arbitrary context that can be adapted with application-specific strategies. The hot-spot that supports this feature is a composition of the Template Method and Strategy patterns, where Context assumes simultaneously the role "Context" of the Strategy pattern and the role "AbstractClass" of Template Method pattern, with addStrategies(Set) as a hook method. Context uses a set of strategies abstracted by the Strategy interface. In order to define the desired strategies in a specialization, we need to develop a subclass of Context that overrides addStrategies(Set). On the other hand, Context class embodies another concern, reflected by the otherConcern() method. We omit the contents of this other concern here and simply refer to it as "other concern". The corresponding hot-spot is an adaptation of the Template Method pattern, having otherConcern() as a hook method.

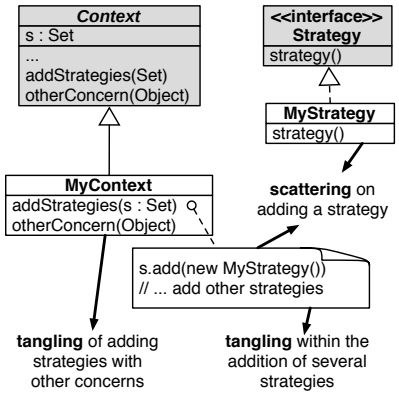

Figure 1. Scattering and tangling in a framework specialization. Darker elements are part of the framework; white elements are classes specific an example specialization.

The specialization presented in Figure 1, consisting of the classes MyContext and MyStrategy, suffers both from scattering and tangling: the inclusion of a strategy involves both MyStrategy and MyContext classes (*heterogeneous crosscut* [9]) and, two different specialization concerns – the inclusion of a strategy and "other concern" – are tangled in the class MyContext. Moreover, if different strategies are considered different specialization concerns, these would be tangled in the method addStrategies(Set) of MyContext.

When reasoning about the bodies of the addStrategies(Set) methods across different specializations, one can realize that they are in a sense redundant, since the only thing that varies is the name of the strategies that are being instantiated (see Figure 2). In this case, we have a *homogeneous crosscut* [9] in inter-specialization scattering. This issue also motivates our approach, hinting us that solutions with more effective reuse can be achieved.

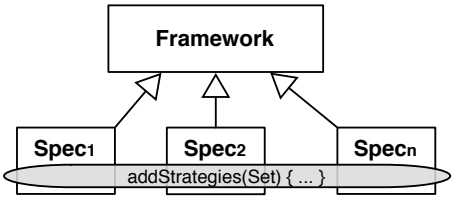

Figure 2. Inter-specialization scattering. Example of an homogeneous crosscut related to the definition of the method addStrategy(Set) across distinct specializations.

On the basis of the analyzed example, we can infer that hot-spots based on unification (e.g. in Context) lead to tangling, whereas hot-spots based on separation (e.g. separation of Context and Strategy) lead to scattering. In the former case, if a class

has hook methods belonging to different hot-spots, its extensions will necessarily be tangled. In the latter case, when a template class is adapted by plugging instances of hook classes, the adaptation of the hot-spot is scattered by the module that adapts the template class and the modules that implement the hook classes.

3. SPECIALIZATION ASPECTS

In the previous section we characterized framework hot-spots and showed that the scattered and tangled nature of framework specializations is directly related to the nature of its hot-spots. The goal of our approach is to eliminate scattering and tangling on the development of framework specializations and support a one-to-one mapping of features to implementation modules.

We propose the concept of *specialization aspect* – an abstract aspect that modularizes a hot-spot, allowing variation to be achieved through the definition of subaspects (i.e. concrete extensions of the abstract aspect). Notice that specialization aspects do not add functionality to the framework. They are just a means of expressing the framework hot-spots that, as we will show, support the development of specializations in a step-wise way.

Figure 3 depicts our approach, distinguishing the roles of domain engineers, which are responsible for the development of the framework plus its specialization aspects, and application engineers, which according to their feature requirements, are responsible for specializing the framework, by developing a set of *application aspects* which are extensions of the specialization aspects.

In this section we address the design and implementation of specialization aspects. Designs are expressed in Theme/UML, a generic notation for aspect design proposed in [8], while implementations are given in AspectJ. Theme/UML supports *design themes*, a UML package with stereotype <<theme>> that allows to describe crosscutting behavior. This package contains template parameters for crosscut elements, which can be bound to concrete elements in a particular context. Although Theme/UML allows to have both classes and operations as parameters, we only make use of class template parameters in this paper.

In the next subsections, we consider again the example introduced in Figure 1 and we progressively show how scattering and tangling can be eliminated from framework specializations through the representation of hot-spots in terms of specialization aspects. Moreover, we consider more complex variants of the initial example, that allow to illustrate the definition of a specialization aspect through refinement and composition of specialization aspects.

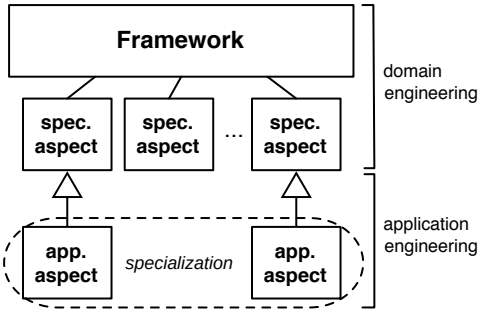

Figure 3. Approach overview.

3.1 De-scattering hot-spots

Consider again the example introduced in the previous section, in particular the hot-spot that supports the inclusion of strategies.

Figure 4 presents a design theme partially describing a specialization aspect concerning the inclusion of a strategy in a specialization of the framework. The theme involves the framework classes

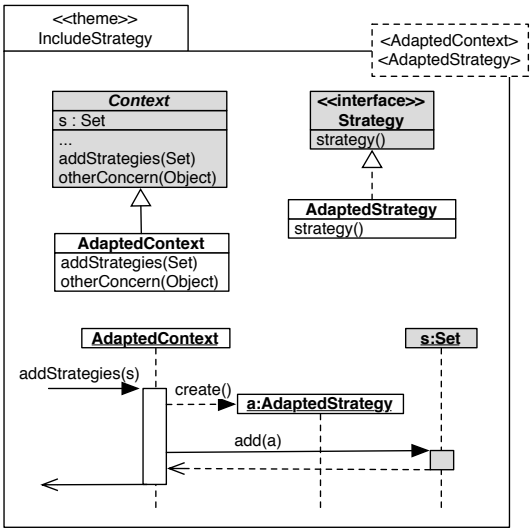

Figure 4. The aspect of including a strategy in a specialization of the framework example presented in Figure 1.

Context and Strategy and two template classes – AdaptedContext and AdaptedStrategy, representing respectively the concrete context and the concrete strategy to be included in that context. The sequence diagram describes the required behavior of addStrategies(Set) in order to implement the inclusion of a strategy.

One could argue that if a specialization is using only black-box components of the framework, such as strategies, there would be no scattering since everything related to including strategies would be located in AdaptedContext. However, there would be tangling in this module, since it also deals with "other concern".

The complete description of the specialization aspect concerning the inclusion of a strategy is given below, in terms of an abstract aspect coded in AspectJ. This implementation is based on the idioms *template advice* and *template pointcut* [17].

```
abstract aspect IncludeStrategy implements Strategy {
  abstract pointcut context();

  after(Set s) :
    execution(void Context.addStrategies(Set)) &&
    args(s) && context() {
      addToSet(this,s);
    }

  void addToSet(Strategy strategy, Set s) {
    s.add(strategy);
  }
}
```

The template parameter AdaptedStrategy of the theme is represented by the abstract aspect IncludeStrategy, which is itself a strategy (implements Strategy), whereas AdaptedContext is represented by the abstract pointcut context(). Specialization is achieved through the definition of a concrete subaspect of IncludeStrategy implementing the strategy() method and the context() pointcut, defining in this way the concrete context and strategy to be used. Notice that IncludeStrategy takes the responsibility for adding the concrete strategy to the set of strategies of the context and, hence, specializations do not have to deal with this issue.

The aspect MyStrategy presented next is an example of a subaspect of IncludeStrategy.

```
aspect MyStrategy extends IncludeStrategy {
  void strategy() {
    System.out.println("This is my strategy");
  }

  pointcut context() : target(MyContext);
}
```

This aspect defines a specialization where the role of context is assumed by the class MyContext (presented below) and the strategy to be used in this context is simply the printing of a message.

Specializations are supposed to define the pointcuts only on their classes/aspects, and never on the framework classes/aspects, in order to keep the separation between framework and specialization functionality.

Notice that in the initial representation of the hot-spot for adding strategies, the method addStrategies(Set) was included as part of Context with the exclusive purpose of allowing specializations to override it, defining which strategies to include. However, since the specialization aspect IncludeStrategy is handling the necessary behavior of addStrategies(Set), this method can now become hidden from the specializations. An interesting effect is that, in order to include strategies, the framework user does not need to know about any details of Context.

```
class MyContext extends Context {
  void otherConcern(Object o) {
    System.out.println("My other concern...");
  }
}
```

This class defines, as expected, that the concrete context is a subclass of the abstract class Context and contains just the method otherConcern(Object) for handling "other concern".

3.2 Un-tangling hot-spots

The previous subsection ended with a solution that de-scattered the inclusion of strategies, modularizing the corresponding hot-spot in IncludeStrategy. However, as seen in the last code snippet, the subclasses of Context still need to have the "other concern" tangled in it.

This subsection shows how the "other concern" can be untangled from Context. The solution proposed here is related to the separation construction principle of hot-spot design. Because we want to separate otherConcern(Object) from Context, we consider a new interface OtherConcern including the otherConcern(Object) operation. Figure 5 describes the desired un-tangled solution as a design theme, having AdaptedContext and AdaptedOther as template parameters. The first parameter represents an arbitrary subclass of Context, while the latter represents our "other concern". The behavior of otherConcern(Object) consists only of the call to AdaptedOther.otherConcern(Object), as if one operation is replaced by the other.

Below, a complete description of the specialization aspect that supports the "other concern" in the framework specializations is presented.

```
abstract aspect HandleOtherConcern {
  abstract void otherConcern(Object o);

  abstract pointcut context();

  void around(Object o):
    execution(void Context.otherConcern(Object)) &&
    args(o) && context() {

      otherConcern(o);
    }
}
```

The mechanisms used in this solution are similar to the ones used in the aspect IncludeStrategy. However, in this case, because the

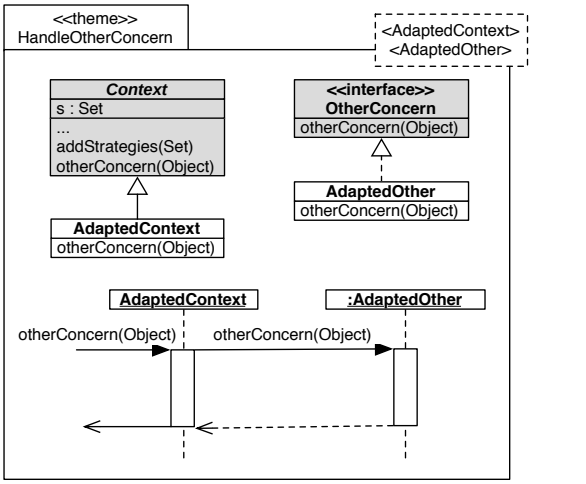

Figure 5. The aspect of "other concern" in a framework specialization.

"other concern" refers to a single operation, it is necessary to use the *around* advice.

The aspect MyOtherConcern presented next is an example of a subaspect of HandleOtherConcern.

```
aspect MyOtherConcern extends HandleOtherConcern {
  void otherConcern(Object o) {
    System.out.println("My other concern...");
  }

  pointcut context() : target(MyContext);
}
```

Analogously to the case explained in the previous subsection, Context.otherConcern(Object) can now become hidden from the specializations, since it has its own specialization aspect. Therefore, the MyContext class can be defined as being just an empty extension of Context.

```
class MyContext extends Context {

}
```

3.3 Refining specialization aspects

So far, we have addressed the design and implementation of independent specialization concerns. However, specialization concerns are not necessarily independent. In this subsection, we address the refinement relationship between specialization aspects and show how the inheritance relationship between abstract aspects can be used to support the refinement of specialization aspects.

Suppose that the initial framework used in Figure 1 provides a default strategy to be used use in a black-box manner. Figure 6 describes a design theme for including this new default strategy (DefaultStrategy). The solution is very similar to the design theme of Figure 4, except that the strategy that has to be added is part of the framework and, therefore, there is a single template parameter (AdaptedContext).

Since we have already provided the specialization aspect IncludeStrategy for including an arbitrary strategy, the implementation of the design theme of Figure 6 can be defined as a refinement of this aspect.

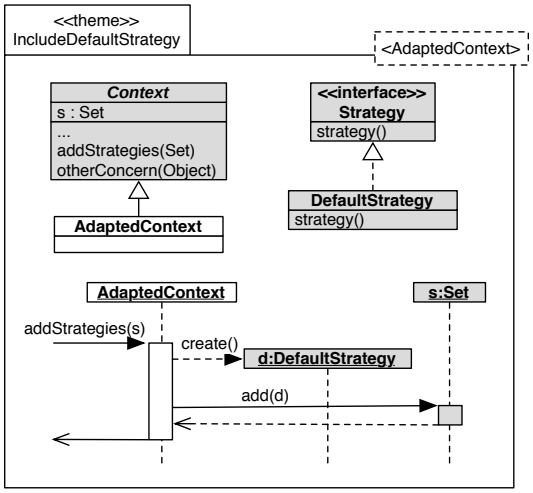

Figure 6. The aspect of including the default strategy in a framework specialization.

```
abstract aspect IncludeDefaultStrategy
  extends IncludeStrategy {

  void strategy() {
    System.out.println("The default strategy");
  }
}
```

It is still an abstract aspect since it not defines context(), but it overrides strategy() for implementing the interface Strategy. Subaspects only need to define the specific context, such as in the following example.

```
aspect DefaultStrategy
  extends IncludeDefaultStrategy {

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

3.4 Composing specialization aspects

As mentioned before, specialization concerns are not necessarily independent. We have found that the two fundamental ways to relate specialization aspects are refinement and composition. This subsection addresses the latter. By composition, we mean a collaboration between specialization aspects that is intended to allow their subaspects to define pointcuts between them. Composition is needed due to the fact that some features depend on other features, i.e. certain features cannot be included in an application without including others, as in the example that we present next.

Suppose that the initial framework used in Figure 1 provides a class StrategyProxy that implements Strategy. Following the *Proxy* design pattern [14], the idea is to allow the wrapping of strategies so that certain additional behavior can be enforced.

Given that we already have IncludeStrategy, a specialization aspect for including strategies, the new specialization concern can rely on IncludeStrategy. For the definition of a specialization aspect supporting the inclusion of a proxy on a strategy we propose a solution based on composition, having IncludeStrategy as a *backend* aspect. Figure 7 presents a design of this solution, where an arbitrary subaspect of IncludeStrategy is represented as a template parameter – AdaptedStrategy. The inclusion of a proxy requires addToSet(Strategy,Set) to be invoked with an instance of StrategyProxy as a parameter, wrapping the parameter of type Strategy. In the theme, we have used the suffix "@normal" to denote the normal behavior of addToSet(Strategy,Set). Notice also that AdaptedStrategy can be either a direct subaspect of IncludeStrategy or an indirect one (e.g. the DefaultStrategy presented previously). We have used the label "..." in the inheritance relationship to denote this transitive closure.

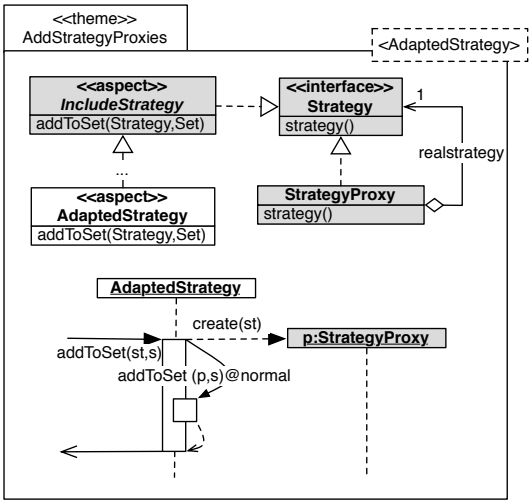

Figure 7. The aspect of adding a proxy on a strategy that was included using IncludeStrategy.

The following is an implementation of the design theme AddStrategyProxies, using similar mechanisms to the previous examples.

```
abstract aspect AddStrategyProxies {
  abstract pointcut strategies();

  void around(Strategy st, Set s) :
    execution(
      void IncludeStrategy.addToSet(Strategy,Set)) &&
      args(st,s) && strategies() {

      proceed(new StrategyProxy(st),s);
    }
}
```

The template parameter AdaptedStrategy is represented by the abstract pointcut strategies(). Nevertheless, this pointcut is intended to support the matching of several strategies. Instead of normal invocations of addToSet(Strategy,Set), the first parameter is wrapped in a StrategyProxy instance and then the method invocation proceeds normally. When specializing the framework, the pointcut strategies() has to be defined on a subaspect of IncludeStrategy, which in the given examples could be MyStrategy or DefaultStrategy. We present below an example that considers both simultaneously.

```
aspect MyProxies extends AddStrategyProxies {
  pointcut strategies() :
    target(MyStrategy) || target(DefaultStrategy);
}
```

Specialization aspects which collaborate have to consider that in their design. In IncludeStrategy, instead of directly adding the strategy to the set, the method addToSet(Strategy,Set) was left on purpose, in order to allow an elegant solution for their composition.

3.5 Other issues

All the given implementations of specialization aspects make use of the template advice idiom, having abstract pointcuts that are intended to be overridden to match a certain quantity of classes/aspects of a certain type. For instance, the pointcut AddStrategyProxies.strategies() can match several subaspects of IncludeStrategy, while the pointcut HandleOtherConcern.context() is

intended to match a single subclass of Context. Moreover, specialization aspects may have an associated constraint on the cardinality of its subaspects. For instance, IncludeStrategy is intended to have several extensions, while HandleOtherConcern is not.

In AspectJ, it does not exist a static mechanism that enforces that a pointcut has a certain type. For instance in the IncludeStrategy specialization aspect, we would like to enforce at static weaving time that the definition of the pointcut context() is of type *target* and matches a subclass of Context. A static mechanism for restricting the number of subaspects would also be useful. Although there is no language support for achieving this, it would not be difficult to have tool support for checking the valid pointcut definitions. In any case, for the given specialization aspects, incorrect definition of these pointcuts just implies that the aspect has no effect, because the advice pointcuts are type-safe. Method signatures are defined against framework types, and no matter which classes specializations point, crosscutting behavior is never going to be introduced at an incorrect location. Nevertheless, since specializations only deal with pointcuts matching exact classes, the *fragile pointcut problem* [39] gets diminished, since the scope of joinpoints is a small subset of AspectJ's joinpoint model.

Another import issue concerns aspect precedences. In our initial example of Figure 1, the Context class contains a set of strategies. Therefore, when having several subaspects of IncludeStrategy, the order in which the strategies are added to the set is not relevant. However, in some situations the definition of aspect precedences is essential. Assuming that this would be the case in our example, we could have an aspect StrategyOrder defining, for instance, that the insertion of MyStrategy is done before DefaultStrategy, setting the first aspect with higher precedence.

```
aspect StrategyOrder {
  declare precedence: DefaultStrategy, MyStrategy;
}
```

An advantage of the proposed solutions is that they are only based on dynamic crosscutting, i.e. the specializations aspects only introduce behavior in the applications without the need of using *introductions* – AspectJ's open class mechanism – which can be considered to weaken *independent extensibility* (e.g. [29]). As another positive characteristic, the solution preserves the *obliviousness principle* of AOP [13], since the framework is not aware of the specialization aspects.

4. EVALUATION

This section evaluates our approach in comparison with traditional object-oriented framework specialization. We first present the traditional way and discuss its characteristics and limitations, in order to point out the benefits (subsection 4.1) and trade-offs (subsection 4.2) of our approach. This evaluation considers the initial framework example with the increments introduced along Section 3, as well as the proposed specialization aspects.

Figure 8 synthesizes the traditional OO solution equivalent to the final solution that was achieved in the previous section – a context MyContext adapted with two strategies, MyStrategy (application-specific) and DefaultStrategy (framework), and "other concern". The two strategies are included with the framework proxy StrategyProxy. The Java code that implements the module MyContext in this solution is also presented, in order to give a clear perception of the limitations of this solution.

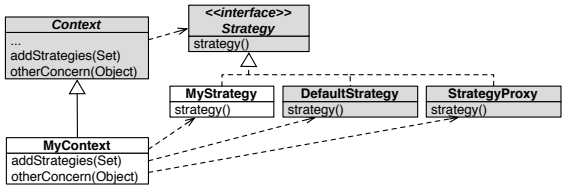

Figure 8. Traditional OO solution.

```
class MyContext extends Context {
  void addStrategies(Set s) {
    s.add(new StrategyProxy(new MyStrategy()));
    s.add(new StrategyProxy(new DefaultStrategy()));
  }

  void otherConcern(Object o) {
    System.out.println("My other concern...");
  }
}
```

Notice that MyContext is dependent on all the other specialization modules plus the framework elements. For instance, adding or removing strategies, as well as adding or removing a proxy, involves modifications in this module. The module clearly suffers from tangling. In frameworks from the real world, analogous situations can have a much larger number of hook methods, which are relevant to several distinct concerns.

Figure 9 synthesizes our solution using specialization aspects. In addition to the framework classes, the specialization aspects are drawn with thicker line and have stereotype <<aspect>>. The stereotyped dependencies (i.e. <<context>>, <<strategy>>) represent the pointcut definitions that are defined in the application aspects according to the examples of the previous section.

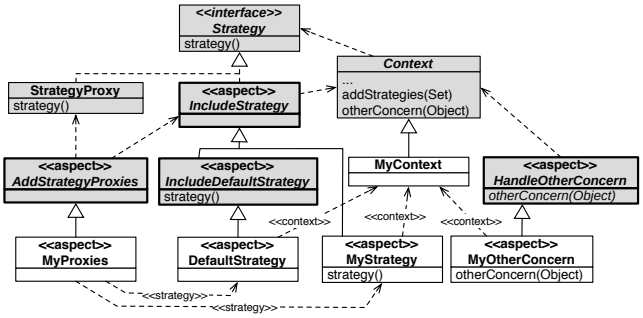

Figure 9. Solution using specialization aspects. These are part of the framework and have stereotype <<aspect>>.

Notice that the framework element DefaultStrategy is not present in our solution; its role is played by IncludeDefaultStrategy. At first glance, this may seem to contradict the conservative nature of our approach, which we argued capable of being applied to a framework "as is". However, the framework design did not change at all. DefaultStrategy is as part of the framework as any other class implementing Strategy could be.

Notice also the inversion of dependencies in the specialization modules – the strategies and the "other concern" now depend on the context. However, these dependencies have a different nature since they are due to the pointcut definitions, which we refer as *pointcut dependencies*. The specialization modules do not have normal dependencies between them, just pointcut dependencies (which can be seen as a static reference). This dependency inversion has interesting characteristics that are discussed in the next subsection.

4.1 Benefits

This subsection points out the benefits of our approach concerning feature cohesion and configurability, cognitive complexity of framework usage, reuse, framework evolution, and specializations evolution.

Feature cohesion and *configurability*. Each specialization module implements one and just one feature, therefore the solution has high feature cohesion. This allows configurability of the specialization, in the sense that application features can vary at weaving time, without the need to modify source code. For instance, removing the default strategy feature from the specialization can simply be achieved by not including the aspect DefaultStrategy in the weaving. The same applies for the proxies and the adaptation of the "other concern". Feature variability at weaving time (i.e. compile time) is not possible in the traditional OO solution. Another issue concerns the addition of features in a step-wise way. Suppose the specialization given as example without the strategy proxies. If we wish to add the feature of having a proxy on a certain strategy, we simply need to develop a subaspect of AddStrategyProxies. This new module implements the desired feature as a non-invasive and independent increment in the specialization. This type of abstraction is also not possible in the traditional OO solution, where invasive modification of MyContext would be necessary.

Cognitive complexity of framework usage. We argue that our approach reduces cognitive complexity when learning and using the framework, due to an abstraction raise to the features level in the development of specializations. Each feature is plugged in a specialization using a single and well identified location – the specialization aspect – having a one-to-one mapping of features supported by the framework to necessary elements for implementing features. The poincut dependencies between specialization modules reflect the relations between the domain concepts. For instance, the relations "a strategy included in a context" and "a proxy for a strategy" are directly mapped to the pointcut definitions that point a context/strategy. For adding a strategy or a proxy there is no need to know about Context internals nor to implement anything in its subclasses, in opposition to the traditional OO solution. These type of issues result in less required documentation for using the framework, as argued with more detail in the next subsection.

Reuse. The homogeneous crosscut of the method addStrategies(Set) of Figure 2 described previously was eliminated, and therefore there are gains in reuse without significant loss of flexibility. Notice in our solution (Figure 9) that the specialization does not deal and does not have to know about addStrategies(Set).

Framework evolution. The evolution of framework classes is more flexible. Consider, for instance, the inclusion of strategies using our approach. If Context modifies the way to include strategies, a modification of specialization aspect IncludeStrategy will be required. However, existing specializations will not be affected, since they only refer a class of type Context. Concerning the addition of framework features, Context can introduce new hook methods with default/empty implementations, and a new specialization aspect can be developed for handling the new concern. In turn, the existing specialization modules would not have to modified, but instead a new aspect for handling the new concern could be developed.

Specializations evolution. Suppose that there was a first version of the framework consisting of the initial example, without the StrategyProxy element, and a second version which includes it. Existing specializations of the first framework version could be upgraded in a non-invasive way, by incrementally developing a subaspect of the AddStrategyProxies aspect, which was included in the second version of the framework. The application developer would not need to know any internals of an existing specialization in order to upgrade it to the second version, it would just be necessary to locate subaspects of IncludeStrategy for adding proxies.

4.2 Trade-off

Our approach requires the implementation of specialization aspects in addition to framework classes. However, the abstraction level is raised, allowing an easier usage by framework users. We found this trade-off advantageous for several reasons:

- A single framework is intended to be used to build several applications. Therefore, the effort shift from framework users to framework developers is likely to be advantageous.
- Relieving the framework users from developing behavioral code for adapting template classes with hook classes (e.g. as in Context), which we argued to be redundant, is likely to reduce unintended (and eventually incorrect) usages of the framework.
- If the framework developers – the ones who know everything about the framework – can write documentation and examples on how to use the it correctly, they should be able to easily write specialization aspects for it.
- Having specialization aspects, the amount of necessary documentation for framework usage is likely to be less and simpler to maintain than the traditional approach. Since the support for developing features is modularized in specialization aspects, the documentation for using it becomes modularized, too. Moreover, since certain hook methods become hidden from specializations, there is no need to provide user documentation about them, explaining how they should be used.

Our approach implies frameworks to have more entities, due to the existence of the specialization aspects. Therefore, one may be concerned with scalability issues when applying the technique to large frameworks. Notice that the number of specialization aspects that a framework should have is directly related with the number of hot-spots, not with the framework size itself. Nevertheless, the complexity of developing and maintaining a set of specialization aspects does not increase along with its cardinality. This is justified by the fact that each specialization aspect handles a certain framework hot-spot, i.e. a typically small set of related classes, remaining independent from the other specialization aspects, except from those that it is intended to be composed with.

5. CASE STUDY

This section presents the results of applying our approach in JHotDraw [38], a GUI framework consisting of 195 classes, with an approximately 27K lines of code[1]. This framework applies several well-known design patterns (e.g. GoF catalog [14]), and it is generally considered to be well designed. This was the main reason for choosing JHotDraw as the framework for evaluating our approach. Since it does not suffer from bad design, we argue that equivalent benefits could not be achieved by having a better designed framework.

This case study stands as a proof of concept of the approach. Conventionally, object-oriented frameworks are specialized by inheritance (white-box) and object composition (black-box). Since several cases involving both ways were covered in the case study, we believe that the technique can be directly applied in conventional frameworks. However, the technique may not be able to be applied in a straightforward and elegant manner in frameworks that were not built for being specialized just by inheritance and object composition. For instance, we refer to cases such as J2EE com-

[1] The implementations of the specialization aspects developed for the case study are available from http://www.cs.tut.fi/∼lealsant/aosd.

ponents or Eclipse plugins, where dynamic class loading mechanisms were adopted, involving necessarily configuration files (e.g. in XML) where information (e.g. class names) must be given by applications (i.e. specializations). However, we argue that in this kind of frameworks, there is always an independent "core", i.e. *"a set of classes that embodies an abstract design"* [22], suitable of having other means for being specialized.

JHotDraw was used "as is", except that certain classes which consist of default framework elements (e.g. figures) were "converted" to aspects, analogously to the case with DefaultStrategy, explained in Section 4.

The specialization aspects for JHotDraw were written in AspectJ. We adopted this technology because it is a seamless aspect-orientation extension to Java, meaning that programming in AspectJ is effectively programming in Java plus aspects. This is convenient due to the conservative nature of our approach. Nevertheless, AspectJ is the most stable AOP language and it its programming primitives broadly match our needs.

JHotDraw deals with technical structured 2D graphics and it is fairly generic. For the purpose of the case study we considered the main extensible parts of the framework, which are depicted in Figure 10. Specializations can adapt the framework in terms of (i) menus, (ii) commands, (iii) figures, (iv) connection figures, (v) tools for creating figures and connections, (vi) valid connections concerning the source and target ends of a connection figure, and (vii) undo of tool actions. Consider also that a specialization can use default elements, such as menus or figures.

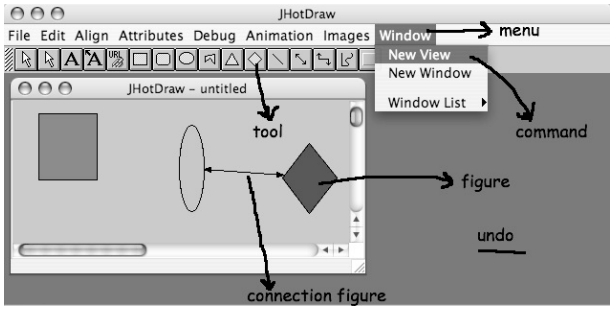

Figure 10. Specialization of the JHotDraw framework – the main extensible parts are marked over the screenshot.

As in framework development, where a stable version of a framework usually results from many design iterations, the development of the specialization aspects for JHotDraw also went through the same process.

The domain engineering activities for enhancing the framework consisted in the following:

1. Establishment of meaningful goals for application engineers, associated with the implementation of application-specific features (e.g. include a *menu*). The criteria for establishing these goals is directly related with the framework domain itself.
2. For each goal obtained in (1), identification of the framework hot-spot that allows its implementation, and development of a specialization aspect for it, which is potentially to be used in composition with another specialization aspects (e.g. include a *command* in a *menu*).

Concerning application engineering, the activities for specializing the framework consisted in the following:

1. According to the desired features in an application, selection of a set of relevant specialization aspects. A selected aspect may be used in more than one feature (e.g. the addition of two *menus*).

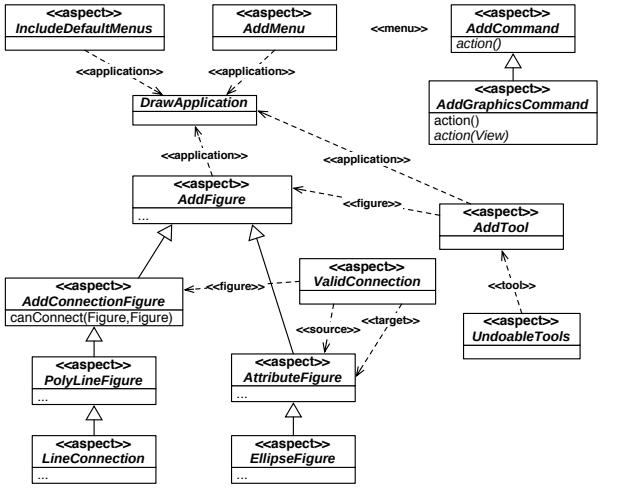

Figure 11. Specialization aspects for the JHotDraw framework.

2. Extension of the aspects selected in (1) in a step-wise way. If aspects are extended according to their dependencies (e.g. aspect for adding a *menu* is developed before the aspect to add a *command*), the effect of each increment (i.e. aspect extension) is immediately visible upon compilation.

Figure 11 presents the specialization aspects that were able to be developed, plus DrawApplication, which is the main class for having an application. Each element with stereotype <<aspect>> corresponds to a specialization aspect, whereas its stereotyped dependencies represent the abstract pointcuts that have to be defined by its subaspects. For instance, if UndoableTools has a dependency <<tool>> to AddTool, means that the subaspects of UndoableTools have to define the pointcut *tool* on an aspect of type AddTool. Consider that subaspects can be indirect, for example, LineConnection is an indirect subaspect of AddConnectionFigure. Analogously, the same applies when the target of these dependencies is a class.

The methods that a specialization has to implement are shown in italics, for instance, the subaspects of AddCommand have to implement action(). In order to keep the illustration simple, we used "..." for denoting the presence of several methods. When there are no methods, the subaspect only has to define the pointcuts, and if case, to parameterize the specialization aspects via its constructor (e.g. AddMenu is parameterized with the menu name).

A specialization aspect may have refinements. For example, AddGraphicsCommand refines AddCommand, implementing action() and introducing another abstract method, action(View), but keeping <<menu>> undefined.

These specialization aspects were developed using the mechanisms described in Section 3, and in terms of AspectJ's joinpoint model, they only rely on exact method pointcut. Some specialization aspects depend on others, but they are designed in such a way that its subaspects do not need to have any dependencies between them, except for the pointcut dependencies. This allows the aspects that constitute a specialization to be independent, and therefore, to be composed transparently.

One can observe in Figure 12 that there are several specialization aspects that have an <<application>> abstract pointcut that is intended to match the type DrawApplication. Since a JHotDraw application is supposed to have a single main class, a global pointcut primitive, such as the one considered in the AspectBench Compiler [3], could be extremely useful. For instance in our example,

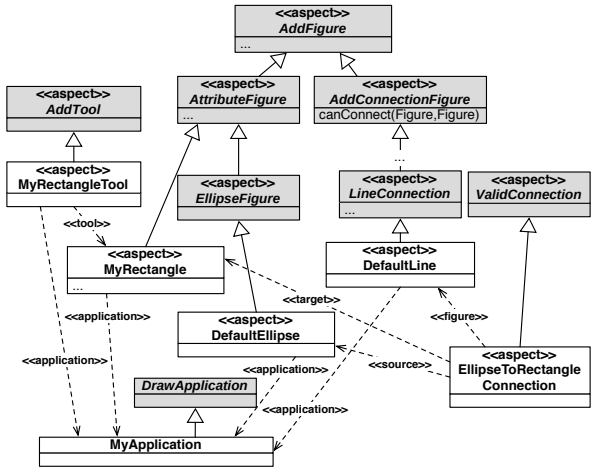

Figure 12. Example of JHotDraw application developed using the specialization aspects (these are represented in darker color).

we could have a global abstract pointcut in a single aspect, where specializations would define their main class.

Figure 12 presents an example of an application developed using the specialization aspects. It consists of a main class (MyApplication), which includes a specific figure (MyRectangle), a default figure (DefaultEllipse), and a default connection (DefaultLine). A tool for creating MyRectangle's is added (MyRectangleTool), as well as a valid connection from DefaultEllipse to MyRectangle (EllipseToRectangleConnection).

In terms of AspectJ's joinpoint model, specializations only have to deal with exact class pointcut, however using both poincut types *target* and *staticinitializition*. Precedences have to be used for guaranteeing certain order relations in the features, such as the left-to-right order of the menus or tools.

Notice that with the given specialization aspects, it is possible for instance to add features without touching the existing modules, such as new figures, new valid connections, undo of certain tools, etc. Notice also that changes in the existing application, such as removing a menu can be done simply by not including the corresponding aspect in the application weaving.

6. RELATED WORK

In this section we review related work, which we divide in (i) approaches that are directly related to ours, (ii) framework learning, (iii) revolutionary approaches, and (iv) domain-specific language approaches.

6.1 Directly related approaches

Reusable aspects in the form of libraries were implemented, in distinct contexts such as GoF patterns [18], concurrency [10], or relationships [34]. Our work aims at introducing aspects on framework-based development, a paradigm with higher level of reuse than libraries.

The work in [32] explores the integration of AOP in object-oriented frameworks, sharing the same objective of using frameworks without modifying them, focusing on C++ frameworks in the context of VLSI CAD applications.

A catalog of idioms for AspectJ is presented in [17]. The specialization aspects presented in our paper make extensive use of the idioms *template advice* and *template pointcut*. For dealing with more complex framework hot-spots, other idioms detailed in the catalog are likely to be useful in our approach.

The AspectBench Compiler [3] proposes powerful AOP primitives, such as the already mentioned *global pointcuts*, which can be exploited in the context of our approach.

We adopted AspectJ as an implementation technology for realizing our approach. However, other AOP languages such as Hyper/J [21] or Caesar [29] could also be adopted. A mapping of Theme/UML *design themes* to Hyper/J is proposed in [8]. Concerning Caesar, since it is also an extension to Java that uses the same joinpoint model than AspectJ, its adoption should be straightforward.

6.2 Framework learning approaches

This subsection presents two examples of approaches based on tool support, which tackle the difficulty of framework usage. These approaches have a conservative nature, since they assume existing framework technology. Their focus is on providing means for developing specializations more easily than just using documentation.

FRED/JavaFrames [16] is a tool capable of providing task-based instantiation of framework specialization patterns [19], based on formal descriptions of these. The tool aids the framework user by guiding and enforcing constraints on the adaptation of hot-spots.

The Strathcona tool [20] is able to recommend framework usage examples, using an existing source code repository. Given a certain framework class, the developer can query the repository for obtaining examples where that class is used.

These kind of approaches address the difficulty of framework learning and usage, but do not raise the abstraction level on their usage – a central goal of our approach.

6.3 Revolutionary approaches

This subsection presents approaches which we consider revolutionary, in a sense that these aim to replace existing software entities as they currently exist, such as the case of frameworks.

A catalog of aspect-oriented refactorings is proposed in [31]. The subject of this work helps on the migration of an existing object-oriented system to an aspect-oriented refactoring.

In [40], aspect-orientation refactoring is studied in the context of middleware software. By "aspectizing" certain features in the implementation of middleware platforms, the authors report that a high degree of configurability can be obtained. This type of feature-level configurability is also an objective of our approach, however, we address configurability of framework specializations, not of frameworks themselves.

AJHotDraw [26] consists of an aspect-oriented refactoring of the JHotDraw framework using AspectJ. The project seems to be focused on the implementation framework internals, rather than on its specialization interface. In case of no changes in the specialization interface, our specialization aspects could apparently be combined with this refactored version of JHotDraw.

Approaches based on feature-oriented programming, such as AHEAD [4], Caesar [30], or aspectual mixin layers (AMLs) [2], propose that software can be built by incremental refinement of implementation modules, allowing high feature cohesion, and therefore powerful configurability of systems in terms of features. Due to this characteristic, these approaches are pointed as effective means for implementing product-lines, i.e. families of related programs.

An evaluation of AOP as a product-line implementation technology is presented in [1]. The authors detail a case study using an hypothetical product-line implemented in AspectJ, where aspects themselves implement part of the product-line features. In our approach, we assume that the product-line (framework) is implemented using conventional object-orientation, while the specialization aspects are an "interface" for developing specializations.

These approaches either imply not reusing existing frameworks, or require major modifications on these. This issue brings the dis-

advantage of "loosing" reliable and proven knowledge on framework construction, besides the frameworks themselves. Nevertheless, revolutionary approaches are likely to have yet unknown drawbacks and problems that can only be realized by large-scale adoption on the software industry.

6.4 Domain-specific language (DSL) approaches

A possible strategy for raising the abstraction level of framework usage is to develop a *domain-specific language* (DSL) for the purpose of specializing it. DSLs naturally allow a considerable abstraction raise when developing applications within the scope of a certain domain.

MetaEdit+ [28] and Software Factories [15] are two examples of approaches supported by commercial tools. These allow to develop domain-specific modeling languages for supporting product derivations of framework-based product-lines.

Domain-specific languages are likely to provide a higher abstraction level than our approach. However, they require considerable investment and are not applicable to domains that evolve rapidly (not stable) or which its applications must have specific functionality that cannot be covered by the DSL.

7. CONCLUSION

In earlier work [37], we proposed AspectJ as a variability mechanism for modularizing framework hot-spots, however with no systematic approach for developing specialization aspects nor case study to support our claim.

In this paper we have explained how framework hot-spots relate to the scattering and tangling phenomena, and we introduced a comprehensive approach for de-scattering and un-tangling hot-spots into specialization aspects. We have focused on the design of specialization aspects, and we introduced the notions of refinement and composition of specialization aspects, demonstrating their usefulness.

We evaluated our approach against the traditional OO solution, concluding that it allows to develop specializations with high feature cohesion, which have benefits on reuse and are flexible in what respects to evolution. Moreover, our approach allows framework specializations to be developed in a step-wise way, facilitating the task of the framework user.

A case study using the JHotDraw framework was described, consisting of a proof of concept for the feasibility of the technique. As discussed in Section 5, the technique that we propose cannot be applied directly in certain types of frameworks, such as in cases where specialization mechanisms based on dynamic class loading were adopted. We consider our approach as an alternative to these types of solutions. Dynamic class loading mechanisms have the disadvantages of not being implementable in plain Java, and of using *reflection*, implying an additional overhead, which may not be suitable for certain type of systems (e.g. mobile phones with limited CPU/memory resources). Moreover, when using dynamic class loading, variation occurs at runtime rather than at compile time, implying that certain compiler checks (e.g. types) can no longer be performed.

The case study on JHotDraw can be considered lightweight. In order to evaluate the technique deeply, for instance, concerning its applicability and scalability, a more extensive case study would have to be carried out.

Frameworks are the most common means to implement product-lines nowadays. Recent case studies point out several difficulties in specializing framework-based product-lines [11], related to product-line architecture learning, inappropriate documentation, and management of variation points. We believe that our approach can be suitable for mitigating these difficulties in existing framework-based product-lines.

Acknowledgments

Thanks to the anonymous reviewers for their useful comments and suggestions to improve the paper.

References

[1] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR*, 2004.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th international conference on software engineering*, 2006.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '04: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, 2005.

[4] 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.

[5] J. Bosch. Product-line architectures in industry: a case study. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, 1999.

[6] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

[7] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. E. Fayad. Framework problems and experiences. In *Building application frameworks: object-oriented foundations of framework design*, chapter 3, pages 55–82. John Wiley and Sons, 1999.

[8] S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In *Aspect-Oriented Software Development*, chapter 19, pages 425–458. Addison-Wesley, 2004.

[9] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004.

[10] C. A. Cunha, J. L. Sobral, and M. P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.

[11] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74:173 – 194, 2005.

[12] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40:32–38, 1997.

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

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools.* John Wiley and Sons, 2005.

[16] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Architecture-oriented programming using fred. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01) (Formal research demo)*, 2001.

[17] 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.

[18] 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.

[19] J. Hautamäki and K. Koskimies. Finding and documenting the specialization interface of an application framework. *Software: Practice and Experience*, (Electronic version):DOI 10.1002/spe.728, 2005.

[20] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[21] IBM. HyperJ. http://www.research.ibm.com/hyperspace/, 2005.

[22] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[24] 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.

[25] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 11:26–49, 1988.

[26] S. E. R. Lab. AJHotDraw. http://ajhotdraw.sourceforge.net/, 2006.

[27] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, 2005.

[28] MetaCase. MetaEdit+ tool. http://www.metacase.com.

[29] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003.

[30] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *ACM Conference on Foundations of Software Engineering (FSE-12)*, 2004.

[31] M. P. Monteiro and J. M. Fernandes. Towards a catalogue of refactorings and code smells for AspectJ. In *T. Aspect-Oriented Software Development I*, 2006.

[32] M. Mortensen and S. Ghosh. Using aspects with object-oriented frameworks. In *AOSD '06: 5th International Conference on Aspect-Oriented Software Development (Industry Track)*, 2006.

[33] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29:45–51, 1996.

[34] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.

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

[36] W. Pree. Hot-spot-driven development. In *Building application frameworks: object-oriented foundations of framework design*, chapter 16, pages 379–394. John Wiley and Sons, 1999.

[37] A. L. Santos, A. Lopes, and K. Koskimies. Modularizing framework hot-spots using aspects. In *Proceedings of the 11th Spanish Conference on Software Engineering and Databases*, 2006.

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

[39] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005.

[40] C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *AOSD '03: Proceedings of the 2nd international conference on aspect-oriented software development*, 2003.