On-Demand Materialization of Aspects for Application Development

Chang Hwan Peter Kim  
The University of Texas at Austin, USA  
chpkim@cs.utexas.edu

Krzysztof Czarnecki  
University of Waterloo, Canada  
czarnecki@acm.org

Don Batory  
The University of Texas at Austin, USA  
batory@cs.utexas.edu

ABSTRACT
Framework-based application development requires applications to be implemented according to rules, recipes and conventions that are documented or assumed by the framework’s Application Programming Interface (API), thereby giving rise to systematic usage patterns. While such usage patterns can be captured cleanly using conventional aspects, their variations, which arise in exceptional conditions, typically cannot be. In this position paper, we propose materializable aspects as a solution to this problem. A materializable aspect behaves as a normal aspect for default joinpoints, but for exceptional joinpoints, it turns into a program transformation and analysis mechanism, with the IDE transforming the advice in-place and allowing the application developer to modify the materialized advice within the semantics of the aspect. We demonstrate the need for materializable aspects through a preliminary study of open-source SWT-based applications and describe our initial implementation of materializable aspects in Eclipse.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Programmer productivity—AOP, materializable aspects, application development, frameworks

Keywords
AOP, materializable aspects, application development, frameworks

1. INTRODUCTION
Object-oriented framework-based application development requires implementing framework-provided concepts by subclassing, implementing interfaces, calling framework services, and so on [2]. These steps involve rules, recipes, and conventions, which we call usage patterns. These usage patterns are difficult to express using the programming language of the framework, such as Java, alone, as they are often crosscutting and require application developers to understand additional objects, classes, interfaces and intricate relationships between them. Previous studies [11, 7] have used aspects to capture crosscutting usage patterns more cleanly. We show an example involving a popular object-oriented GUI framework, Swing.

A notable feature of the Swing framework is its support for configuration of look and feel (L&F) of its applications. An application may change its L&F after startup by calling UIManager.setLookAndFeel(..). However, for the change to take effect, the application must call SwingUtilities.updateComponentTreeUI(Component) afterwards, providing the top-most Component whose subcomponents should reflect the change [10]. By convention, in many applications, whenever a Window changes its L&F, the L&F of that Window is updated immediately afterwards. We could capture this usage pattern using the LookAndFeel aspect in Figure 1.

Figure 1: LookAndFeel aspect

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
Copyright 2008 ACM 978-1-60558-144-6/08/0003 ...$5.00.
public class MessengerApplication extends JFrame {
    public void applySettings(Settings settings) {
        UIManager.setLookAndFeel(settings.getLF());
        if (settings.isRestartDesired())
            restart();
        /* SwingUtilities.updateComponentTreeUI(this); */
    }
}

public void test() {
    UIManager.setLookAndFeel(DefaultLookAndFeel); // test some other settings
    ...
}

public abstract aspect LookAndFeel

Figure 2: MessengerApplication

public abstract aspect LookAndFeel

Figure 3: Abstract LookAndFeel aspect

only in the components that do not require reloading.

The LookAndFeel aspect in Figure 1 cannot accommodate both the default and the exceptional cases. There are several ways of capturing both cases. An intuitive possibility is to make the LookAndFeel aspect in Figure 1 abstract and define a separate subsaspect for each case. The abstract aspect, shown in Figure 3, defines the pointcut change and the advice statement updateWindow that are common to all the subsaspects.

With the abstract aspect in place, we define the aspect for default cases by extending the LookAndFeel abstract aspect as DefaultLookAndFeel aspect and excluding the exceptional joinpoint in Figure 2, line 5, from the pointcut as shown in Figure 4. Then we define another aspect just for the exceptional joinpoint, as shown in Figure 5. In Figure 5, line 6, we pick out the exceptional method call using withincode and then in line 8, pick out settings object by selecting the method MessengerApplication.applySettings through cflow and accessing its arguments. 1 Luckily, we are able to pick out settings object because it is available as a method parameter; had it been a local variable in MessengerApplication.applySettings, it would have been much more cumbersome to write. Also, to be able to call MessengerApplication.restart() (Figure 5, line 15), we have to make the aspect privileged since the method is private.

Multiple aspects are required using conventional AOP: one for default cases, one for each exceptional case, and one for an abstract aspect to capture the commonality in the default and the exceptional aspects. Each exceptional aspect can be considered to be degenerate, as it does not quantify over multiple joinpoints and is practically glued to the base, which makes it awkward to specify and brittle with respect to evolution of the base code.

Based on this example, we make two hypotheses: a problem hypothesis and a solution hypothesis. Our problem hypothesis is that there will be usage patterns with a significant number of exceptions as well as defaults in existing framework-based applications. Although we cannot weave an advice homogeneously for exceptional cases, we believe that we can still provide useful automated software engineering support for them, including code assist, traceability, and program analysis. From this reasoning, our solution hypothesis follows: such support for exceptions and homogeneous weaving for defaults can be provided using the same aspect that is interpreted in an unconventional way. In the following sections, we develop the concept of materializable aspects to support these hypotheses.

2. MATERIALIZABLE ASPECTS

A materializable aspect behaves like a conventional aspect by default, but in exceptional cases, it switches to a program transformation and analysis mechanism that is enabled by an Integrated Development Environment (IDE). Figure 6 shows a materializable aspect in action for the running example introduced in the previous section.

The LookAndFeel framework aspect in Figure 1 stays as is; we don’t need to convert it into Figures 3, 4, and 5 as we had to in the previous section. In Figure 6(a), the LookAndFeel aspect advises lines 6 and 13, as indicated by arrow-shaped joinpoint-markers. Since line 6 is an exceptional joinpoint for which the advice must be woven not necessarily immedi-

1We use cflow only for the purpose of picking out settings object, not for picking out other L&F changes in the control flow of applySettings. This is necessary in AspectJ as withincode doesn’t combine with args.
1. Developer marks exceptional joinpoint.

2. IDE transforms advice in-place.

3. Developer moves in-place advice further down.

4. Developer removes in-place advice. IDE complains.

Figure 6: LookAndFeel materializable aspect in action

a) 1. Developer marks exceptional joinpoint.

```java
public class MessengerApplication extends JFrame {
  public void applySettings(Settings settings) {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    SwingUtilities.updateComponentTreeUI(this);
  }
}
```

b) 2. IDE transforms advice in-place.

```java
public class MessengerApplication extends JFrame {
  public void applySettings(Settings settings) {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    SwingUtilities.updateComponentTreeUI(this);
  }
}
```

c) 3. Developer moves in-place advice further down.

d) 4. Developer removes in-place advice. IDE complains.

2.1 On-demand customization

Both the default and exceptional cases are handled using a single aspect. Advice is woven immediately after for default cases by the compiler and for exceptional cases, they are materialized by the IDE and allowed to be customized within the semantics of the aspect (e.g. sometime after) on-demand by the application developer. Unlike typical generative approaches, code produced must be customization-friendly, i.e. understandable and maintainable. Materializable aspects integrate conventional aspects and program transformation and analysis in a practical way.

2.2 Specification

An aspect is a specification of a temporal relationship between the joinpoints and the advice. For each materialization, the specification imposes two kinds of constraints: reachability and joinpoint context identity by reference or equality by value. For example, for advice, not only must the advice be reachable from the joinpoint, but also, objects or values at the materialized advice must be identical or equal to those picked out at the joinpoint. Although joinpoint context identity or equality is not much of a problem in this case since the caller in line 9 is most likely to represent the caller in line 6, in other cases, for example, when an object is picked out using `target` or `args`, the reference at the materialized advice must represent the same object.

Program analyses will become especially difficult if the materialized joinpoint and advice are allowed to be in different methods. This semantic specification idea can be generalized to other kinds of aspects, including those with `before` and `around` advices.
2.3 Separation between framework and application

Materializable aspects challenge the traditional notion of framework code and application code having to be physically separated. With conventional framework aspects, the framework code (advice) is separated physically from the application code (base). However, situations arise when the framework code becomes inseparable from the application code (base). For example, in Figure 2, the update in apply-Settings is a special kind that requires a restart for a full update and no restart for a partial update. Although the framework code (the update statement in line 8) may make sense alone, the application code (the restart statements in lines 6 and 7) does not make sense without the framework code. In essence, what we have is a joinpoint-specific advice that, although can be separated physically from the joinpoint, the effort required to do so doesn’t seem to be offset by the benefit gained, as shown in Section 1. However, using materializable aspects, although the framework code and application code are physically together, we can still identify the boundary between them and reason about them separately.

3. STUDY OF ASPECTIZABLE USAGE PATTERNS

We conducted a study supporting the problem hypothesis, namely, that usage patterns typically have both defaults and exceptions and that technologies like materializable aspects are needed. In the future, we plan to conduct a follow-up study covering more frameworks and applications that supports the solution hypothesis, namely, that materializable aspects provide a practical solution to the problem.

SWT (and JFace) framework was selected because it is a widely used framework and many applications using it are readily available as open source. 18 open-source applications written against SWT framework were downloaded from SourceForge [13] and studied. We identified three usage patterns that could be expressed by framework aspects by looking for related method calls and objects using an aspect-mining tool [8] and by consulting SWT API documentation [5]. The results are summarized in Table 1. The first column lists the 18 applications and their numbers of downloads as of January 15, 2008. The rest of the columns lists the usage patterns, with each entry having the form {defaults, exceptions} except. The last row of the table shows the number of applications demonstrating the usage pattern, the total number of defaults and exceptions, the total percentage of exceptions and the percentage of exceptions per application. We briefly explain each usage pattern (column).

ShellEventLoop. By default, in 70% of the time across 11 applications, after a Shell (window) is opened, the GUI application must process the events in the application queue repeatedly until the application is terminated. The remaining 30% represent exceptions, including a situation where code (e.g. to open a secondary window) is inserted in between the joinpoint and the advice and a situation where exception handling logic is inserted inside the loop.

ShellSize. By default, in 51% of the time across 13 applications, before a Shell is opened, its size is set to the preferred size determined by its layout manager using pack(). The remaining 49% represent exceptions, including situations where size is set using other methods such as set-Size(...) and setMaximized(true).

ContextMenu. By default, in 67% of the time across 4 applications, after a menu manager creates a context menu from a widget, the context menu is set on the widget. The remaining 33% represent exceptions, including situations where the created context menu is cached for some time before being set on a widget.

Our study shows that there is a good mix of both defaults and exceptions to usage patterns that can be expressed using AspectJ aspects. Although the study revealed only three usage patterns, we believe that if we expanded the notion of aspects beyond those of AspectJ, for example, to those that are less homogeneous and more expressive (see Section 5), then more default cases could arise and in turn, more exceptional cases. In the future, we plan to actually express the usage patterns using materializable aspects and determine the increase in programmer productivity over expressing the usage patterns conventionally.

4. PROTOTYPE IMPLEMENTATION AND REMAINING TECHNICAL CHALLENGES

As we mentioned earlier, a prototype implementing materializable aspects was developed for the Eclipse platform using AspectJ Development Toolkit (AJDT). Only aspects with call pointcuts and after relationships are currently supported. When the application developer annotates a joinpoint (Figure 6(a), line 5) and refreshes the editor, the IDE, underneath the cover, transforms the framework aspect (Figure 1) into an aspect that excludes the joinpoint (Figure 6(a), line 5) and refreshes the editor, the IDE, underneath the cover, transforms the framework aspect (Figure 1) into an aspect that excludes the joinpoint (Figure 4) and the advice is transformed in-place (Figure 6(b), line 7). Unfortunately, as AspectJ cannot exclude particular AST nodes, if identical method calls in the same method, annotating one method call will exclude all the method calls. After the developer customizes the in-place

<table>
<thead>
<tr>
<th>Application</th>
<th>ShellSize</th>
<th>EventLoop</th>
<th>Settings</th>
<th>ContextMenu</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlinkReader</td>
<td>15/11</td>
<td>25%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>DefaultWebEditor</td>
<td>11/32</td>
<td>35.7%</td>
<td>11/22</td>
<td>29%</td>
</tr>
<tr>
<td>EssentialBudget</td>
<td>11/4</td>
<td>25%</td>
<td>10/1</td>
<td>25%</td>
</tr>
<tr>
<td>EssentialCalendar</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>FileNumber</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Full</td>
<td>(6, 4)</td>
<td>100%</td>
<td>(4, 2)</td>
<td>33.3%</td>
</tr>
<tr>
<td>GtTest</td>
<td>10/20</td>
<td>50%</td>
<td>10/20</td>
<td>50%</td>
</tr>
<tr>
<td>GtTest</td>
<td>10/20</td>
<td>50%</td>
<td>10/20</td>
<td>50%</td>
</tr>
<tr>
<td>H oppy</td>
<td>(1, 5)</td>
<td>20%</td>
<td>(2, 5)</td>
<td>20%</td>
</tr>
<tr>
<td>JavaHatching</td>
<td>(4, 13)</td>
<td>25%</td>
<td>(2, 4)</td>
<td>25%</td>
</tr>
<tr>
<td>KeyPad</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>KeyPad</td>
<td>(0, 1)</td>
<td>0%</td>
<td>(0, 1)</td>
<td>0%</td>
</tr>
<tr>
<td>Language</td>
<td>(734, 203)</td>
<td>100%</td>
<td>(6, 5)</td>
<td>100%</td>
</tr>
<tr>
<td>Language</td>
<td>(9, 450)</td>
<td>100%</td>
<td>(6, 5)</td>
<td>100%</td>
</tr>
<tr>
<td>SWTCalendar</td>
<td>(13, 951)</td>
<td>10%</td>
<td>(3, 9)</td>
<td>10%</td>
</tr>
<tr>
<td>VistaDouble</td>
<td>(29, 6)</td>
<td>100%</td>
<td>(30, 6)</td>
<td>100%</td>
</tr>
<tr>
<td>Virgo</td>
<td>(308, 507)</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Virgo</td>
<td>(18, 2)</td>
<td>100%</td>
<td>(18, 2)</td>
<td>100%</td>
</tr>
<tr>
<td>18 apps</td>
<td>15 apps</td>
<td>100%</td>
<td>15 apps</td>
<td>100%</td>
</tr>
<tr>
<td>4 apps</td>
<td>4 apps</td>
<td>100%</td>
<td>4 apps</td>
<td>100%</td>
</tr>
<tr>
<td>37.5% except./app</td>
<td>37.5% except./app</td>
<td>70% except./app</td>
<td>33.3% except./app</td>
<td>33.3% except./app</td>
</tr>
</tbody>
</table>

Table 1: Results of aspectizing SWT usage patterns
advice (Figure 6(c), line 9) and refreshes the editor, interprocedural reachability analysis checks that the materialized advice is always, sometimes, or never reachable from the joinpoint (Figure 6(d)). Context passing from joinpoint to advice is not implemented, which means that advice must be static and that dynamic constructs like cflow are not supported.

The exercise of developing this simple prototype was valuable not only for demonstrating core ideas behind the proposal, but also for understanding its technical challenges and tradeoffs.

4.1 Expressiveness

A more expressive aspect language is needed to pick out precise elements like AST nodes and to express a rich temporal relationship between the joinpoints and the advice. Although such a language will make further (e.g., interprocedural) program analysis and transformation more difficult to implement, it seems that approximations can be used to reduce implementation effort and still allow many usage patterns to be covered.

Arguably, if we push the expressiveness of the aspect language enough, we could arrive at an aspect language that could capture both defaults and exceptions concisely. However, materializable aspects offer an elegant and practical crossover between a language-oriented and a tool-oriented solution that can be implemented and used adequately at a cost fraction of required to implement and learn such an expressive aspect language. Balancing language expressiveness and program transformation and analysis capability is the key to making materializable aspects useful.

4.2 Tool usability

Materializable aspects are tool-centric and there are several interesting issues to be addressed. For example, the IDE may offer different modes of operation for materializable aspects. In interactive mode (e.g., code assist), speed of program analysis and transformation may be important, while in batch mode (e.g., compilation), accuracy and precision may be more important. Also, if materializations occur across multiple joinpoints, the IDE may provide a view through which the application developer can systematically review them and make customizations, rather than having to manually search for all the materializations.

4.3 Framework and application co-evolution

Materializable aspects are part of the framework and the base that they advise and customizations made to materializations are part of the application. Both materializable aspects and their materializations can change, potentially causing inconsistency. For framework-based application development, it is more likely that materializations, not materializable aspects, will change because an application developer typically does not change framework code such as SWT library code (unless he needs to do so locally). But framework and application co-evolution is possible and although it is not the focus of this paper, we give some ideas on how it can be addressed. If the application developer changes or removes a joinpoint so that an aspect no longer applies to it, the materialized advice should be removed or the application developer must be informed that it is no longer necessary. If a pointcut is changed, we remove materializations that no longer apply or inform the application developer. If an advice is changed, we replace the materialized advice occurrences with the new advice or inform the application developer.

5. RELATED WORK

Frameworks. Kulesza et al. [7] propose that applications define aspects to advise extension joinpoints, unlike our approach, which proposes that frameworks define aspects to advise application code. For white-box framework-based application development, which largely involves implementing hook methods, their work is more suitable than our work. However, for black-box framework-based application development, which mainly involves creating objects, wiring them, and calling methods according to behavioural framework rules or conventions, our approach is more suitable than their approach.

With framework specialization aspects [11], an abstract aspect defines a general framework rule, such as a design pattern, with extension points called hot-spots and concrete aspects must fill these hot-spots for each instantiation of the rule. Like the previous related work, framework specialization aspects advise framework, not application, code, making them more suitable for white-box rather than black-box framework-based application development. Also, with framework specialization aspects, when customization is required beyond the pre-planned extension points, one will run into the same problem presented in Section 1, where materialization is more natural than aspect extension.

Generative approaches. Smith treats an aspect as an equivalence relation [12] for which maintenance code is generated when the relation is disturbed. Although materializable aspects are also constraints that generate code, code is generated and customized on-demand, which brings our work closer to multi-level customization [3], which is the idea that intermediate abstractions, like materializable aspects, be introduced to facilitate application development.

A Framework-Specific Modeling Language (FSML) [2] is a systematic approach to modeling framework concepts. Although both FSMLs and materializable aspects are technologies enabling framework-based application development, FSMLs focus on enabling model-driven development, while materializable aspects focus on integrating conventional AOP and program transformation and analysis. Future work will involve seeing how materializable aspects and FSMLs can be integrated.

Expressive aspect languages. An important future work item is to understand how more expressive aspect languages, such as those capable of specifying history of events [6, 4, 1] and data flow [9] and generating aspects [14], can be used as a basis for materializable aspects in a way that will balance language and program transformation and analysis benefits as discussed in Section 4.1.

6. CONCLUSION

Using conventional aspects alone, it is difficult to capture both defaults and exceptions of usage patterns. Materializable aspects, which behave as conventional aspects for default joinpoints and as program transformation and analysis mechanisms for exceptional joinpoints, were presented as a solution, along with a prototype implementation of some of its ideas. The problem hypothesis was supported through a preliminary study of 18 open-source SWT-based applica-
tions, which identified three usage patterns having a balanced mix of both defaults and exceptions. In the future, the solution hypothesis will be tested by widening the scope of aspects to make them more applicable, by studying other frameworks, and by determining the benefits that materializable aspects provide.

7. REFERENCES


