Software Engineering Properties of Languages and Aspect Technologies (SPLAT), 2008

On-Demand Materialization of Aspects for Application Development

Chang Hwan Peter Kim
The University of Texas at Austin, USA
Austin, TX 78712
chpkim@cs.utexas.edu

Krzysztof Czarnecki
University of Waterloo, Canada
Waterloo, ON N2L3G1
czarnecki@acm.org

Don Batory
The University of Texas at Austin, USA
Austin, TX 78712
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, materalizable 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.

```
1 public aspect LookAndFeel {
2    pointcut change(Window window):
3       call(void UIManager.setLookAndFeel(..)) &&
4          within(Window+) && this(window);
5
6    after(Window window) returning: change(window) {
7       SwingUtilities.updateComponentTreeUI(window);
8    }
9 }
```

Figure 1: LookAndFeel aspect

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.

However, there are exceptions to this pattern. For example, consider the class `MessengerApplication` in Figure 2, which is an instant messenger application that changes its L&F in lines 5 and 13. Since `MessengerApplication` is an indirect subclass of `Window`, both calls to `setLookAndFeel` in the class will be advised by the `LookAndFeel` aspect. Although the application developer wants the advice to apply immediately after line 13, he or she does not want the advice to apply immediately after line 5, but wants it to apply immediately after the application-specific code, i.e. lines 6 to 7 involving `restart()`. Where the advice *should* be woven for this exceptional case, but is not, is explicitly shown in line 8. The rationale for the exceptional case is as follows. Some graphical components that are running already have to be reloaded for them to reflect the new L&F setting. Because it is difficult to identify and reload components individually, the entire application has to be restarted for these components to reflect the change. For convenience, the user may choose to restart at a later time and reflect the L&F change

SPLAT 2008, March 31, 2008, Brussels, Belgium.

```
1  public class MessengerApplication extends JFrame
2  {
3     public void applySettings(Settings settings)
4     {
5        UIManager.setLookAndFeel(settings.getLF());
6        if(settings.isRestartDesired())
7           restart();
8        /* SwingUtilities.updateComponentTreeUI(this); */
9     }
10
11    public void test()
12    {
13       UIManager.setLookAndFeel(lookAndFeel);
14       // test some other settings
15    }
16    ...
17 }
```

Figure 2: MessengerApplication

```
1  public abstract aspect LookAndFeel
2  {
3     pointcut change(Window window):
4        call(void UIManager.setLookAndFeel(..)) &&
5           within(Window+) && this(window);
6
7     public void updateWindow(Window window)
8     {
9        SwingUtilities.updateComponentTreeUI(window);
10    }
11 }
```

Figure 3: Abstract LookAndFeel aspect

```
1  public aspect DefaultLookAndFeel extends LookAndFeel
2  {
3     pointcut defaultChange(Window window):
4        change(window) && !withincode(void
5           MessengerApplication.applySettings(Settings));
6
7     after(Window window) returning: defaultChange(window)
8     {
9        updateWindow(window);
10    }
11 }
```

Figure 4: DefaultLookAndFeel aspect

```
1  privileged aspect MessengerLookAndFeel extends LookAndFeel
2  {
3     pointcut messengerChange(Window window, Settings settings):
4        change(window) &&
5          withincode(void MessengerApplication.
6              applySettings(Settings)) &&
7          cflow(call(void MessengerApplication.
8              applySettings(Settings)) && args(settings));
9
10    after(Window window, Settings settings) returning:
11       messengerChange(window, settings) {
12       MessengerApplication application =
13                (MessengerApplication)window;
14       if(settings.isRestartDesired())
15          application.restart();
16       updateWindow(application);
17    }
18 }
```

Figure 5: MessengerLookAndFeel aspect

only in the components that do not require reloading.

The `LookAndFeel` aspect in Figure 1 cannot accomodate 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 subaspect 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 subaspects.

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.

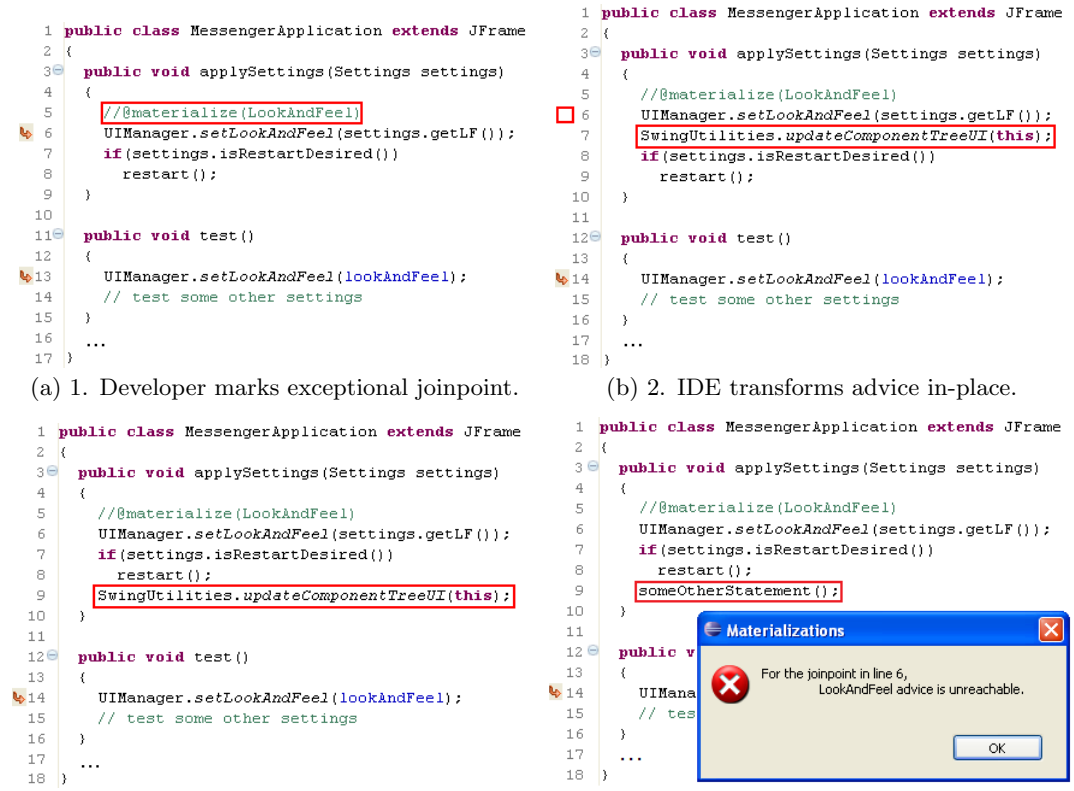
Based on this example, we make two hypotheses: a problem hypothesis and a solution hypothesis. Our problem hypothesis is that there will be usage patterns with a significant number of exceptions as well as defaults in existing framework-based applications. Although we cannot weave an advice homogeneously 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-

[1] We 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`.



(a) 1. Developer marks exceptional joinpoint.

(b) 2. IDE transforms advice in-place.

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

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

Figure 6: LookAndFeel materializable aspect in action

ately after, but after the if-statement, the application developer places an annotation above the joinpoint and refreshes the editor. Then, as shown in Figure 6(b), the IDE processes the annotation and transforms the advice in-place in line 7. Note that the joinpoint-marker is no longer on line 6, but is still on line 14. Then, as shown in Figure 6(c), the application developer moves the in-place advice further down to line 9, where it should be. For each exceptional joinpoint, the IDE checks to see if the *after* relationship between the joinpoint and the in-place advice specified by the aspect is satisfied. *after* obviously has a wide meaning. For now, we mean that the advice must be reachable from a joinpoint by the end of the control flow of the method in which the joinpoint shadow appears. So if the `LookAndFeel` advice were replaced by some other statement (and the advice is not executed in its control flow) as shown in Figure 6(d), the IDE would complain that the advice is not reachable from the joinpoint.

It is important to note that using materializable aspects, we can capture both the default and the exceptional cases uniformly using the original framework aspect (Figure 1) and not three degenerate application-specific aspects (Figures 3, 4, and 5).

We discuss several key ideas behind materializable aspects that we have implemented in a prototype tool.

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 *after* 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 Figure 6(c) since `this` 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. For example, in Figure 2, the update in `applySettings` 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} %exceptions`. 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 `setSize(..)` and `setMaximized(true)`.

Table 1: Results of aspectizing SWT usage patterns

	Shell EventLoop	ShellSize	Context Menu
BlogCaster (692)	{3, 1} 25%	{0, 5} 100%	
Deinonychus (4,036)	{1, 6} 85.7%	{5, 2} 29%	
EssentialBudget (7,692)	{1, 0} 0%	{0, 1} 100%	
FaceIt (957)	{3, 0} 0%	{0, 3} 100%	{2, 0} 0%
FileBunker (7,960)			{2, 0} 0%
FnR (17,782)		{0, 4} 100%	{4, 2} 33.3%
GFace (10,209)			
Hopy (1,507)			{0, 2} 100%
JavaHexEditor (9,131)	{1, 3} 75%	{2, 2} 50%	
PaperClips (8,116)			
CalypsoRCP (6,239)	{3, 0} 0%	{0, 3} 100%	
PinkPotato (n/a)			
RSSOWL (743,203)	{0, 5} 100%	{0, 5} 100%	
SolVE (9,450)	{0, 3} 100%	{0, 3} 100%	
SWTCalendar (13,951)	{3, 0} 0%	{3, 0} 0%	
TuxGuitar (308,907)	{29, 6} 17.1%	{30, 6} 16.7%	
VirgoFTP (20,246)	{18, 2} 10%	{5, 9} 64.3%	
18 apps	11 apps {62, 26} 30% except. 37.5% except./app	13 apps {46, 44} 48.9% except. 70% except./app	4 apps {8, 4} 33.3% except. 33.3% except./app

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 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 exist in the same method, annotating one method call will exclude all the method calls. After the developer customizes the in-place

advice (Figure 6(c), line 9) and refreshes the editor, intra-procedural 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 advices 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. inter-procedural) 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 fraction of the cost 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

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA*, pages 345–364, 2005.

[2] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, pages 692–706, 2006.

[3] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim. Multi-level customization in application engineering. *Commun. ACM*, 49(12):60–65, 2006.

[4] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In M. Aksit, S. Clarke, T. Elrad, and R. E. Filman, editors, *Aspect-Oriented Software Development*, pages 201–218. Addison-Wesley, 2004.

[5] Eclipse. SWT documentation. Document available at `http://http://www.eclipse.org/swt/docs.php`.

[6] R. Filman and K. Havelund. Realizing aspects by transforming for events, 2002.

[7] U. Kulesza, V. Alves, A. F. Garcia, C. J. P. de Lucena, and P. Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *ICSR*, pages 231–245, 2006.

[8] M. Marin, L. Moonen, and A. van Deursen. Fint: Tool support for aspect mining. In *WCRE*, pages 299–300, 2006.

[9] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS 03: Asian Symposium on Programming Languages and Systems*, Beijing, 2003. Springer Verlag.

[10] S. Microsystems. How to set the look and feel. in the java tutorials. Document available at `http://java.sun.com/docs/books/tutorial/uiswing/lookandfeel/plaf.html`.

[11] A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 14–24, New York, NY, USA, 2007. ACM Press.

[12] D. R. Smith. A generative approach to aspect-oriented programming. In *GPCE*, pages 39–54, 2004.

[13] SourceForge. Open-source application repository. `www.sourceforge.net`.

[14] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with meta-AspectJ. In *Generative Programming and Component Engineering (GPCE)*, pages 1–18. Springer-Verlag, October 2004.