Computer Science Conference Presentations, Posters and Proceedings

Computer Science

3-25-2012

An exploratory study of the design impact of language features for aspect-oriented interfaces

Robert Dyer
Iowa State University

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Yuanfang Cai
Drexel University

Recommended Citation

Dyer, Robert; Rajan, Hridesh; and Cai, Yuanfang, "An exploratory study of the design impact of language features for aspect-oriented interfaces" (2012). *Computer Science Conference Presentations, Posters and Proceedings*. 41.
http://lib.dr.iastate.edu/cs_conf/41

An exploratory study of the design impact of language features for aspect-oriented interfaces

Abstract

A variety of language features to modularize crosscutting concerns have recently been discussed, e.g. open modules, annotation-based pointcuts, explicit join points, and quantified-typed events. All of these ideas are essentially a form of aspect-oriented interface between object-oriented and crosscutting modules, but the representation of this in-terface differs.

Disciplines

Computer Sciences | Programming Languages and Compilers | Software Engineering

Comments

This article is published as Dyer, Robert, Hridesh Rajan, and Yuanfang Cai. "An exploratory study of the design impact of language features for aspect-oriented interfaces." In Proceedings of the 11th annual international conference on Aspect-oriented Software Development, pp. 143-154. ACM, 2012. doi: 10.1145/2162049.2162067. Posted with permission.

Rights

An Exploratory Study of the Design Impact of Language Features for Aspect-oriented Interfaces

Robert Dyer Hridesh Rajan
Iowa State University
{rdyer,hridesh}@iastate.edu

Yuanfang Cai
Drexel University
yfcai@cs.drexel.edu

Abstract

A variety of language features to modularize crosscutting concerns have recently been discussed, e.g. open modules, annotation-based pointcuts, explicit join points, and quantified-typed events. All of these ideas are essentially a form of aspect-oriented interface between object-oriented and crosscutting modules, but the representation of this interface differs. While previous works have studied maintenance of AO programs versus OO programs, an empirical comparison of different AO interfaces to each other to investigate their benefits has not been performed. The main contribution of this work is a rigorous empirical study that evaluates the effectiveness of these proposals for AO interfaces towards software maintenance by applying them to 35 different releases of a software product line called MobileMedia and 50 different releases of a web application called Health Watcher. Our comparative analysis using quantitative metrics proposed by Chidamber and Kemerer shows the strengths and weaknesses of these AO interface proposals. Our change impact analysis shows the design stability provided by each of these recent proposals for AO interfaces.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Control structures

General Terms Design, Human Factors, Languages

Keywords aspect-oriented, implicit invocation, empirical study, AO interfaces, annotations, events, open modules

1. Introduction

There has been a large body of recent case studies on the software engineering (SE) benefits of aspect-orientation [7, 8, 12, 13, 16]. These works compute standard SE metrics such as coupling and cohesion and compare aspect-oriented (AO) designs to object-oriented (OO) designs or use the metrics to determine stability and fault-proneness of the systems. However, most of these works focus on comparing AspectJ [15] to Java and do not compare different AO interfaces with each other, leaving developers to wonder about the benefits of one proposal over others.

This work fills that gap. It studies and compares different proposals for aspect-oriented interfaces to study how they impact code changes. For this, we consider a software product line for handling multimedia on mobile devices, called MobileMedia [8] and a web-based health application, called Health Watcher [16, 20]. Similar to previous in-depth analyses by Figueiredo *et al.* [8] and Greenwood *et al.* [11], we present metrics such as coupling and cohesion as well as an analysis of the change propagation across releases. However, unlike those studies we consider not only OO and pattern-based pointcuts (PCD) but also three other proposals for AO interfaces: open modules [1] (OM), annotation-based pointcuts [13] (@PCD), and quantified, typed events [19] (EVT).

Results and Contributions There were several interesting results to come out of our case study. First, the annotation-based pointcut and quantified, typed event approaches showed several benefits, in terms of change impact, over the standard pattern-based pointcut approach.

- The @PCD releases have 18% fewer changed pointcuts than the PCD releases, due to a lack of fragile pointcuts.
- The total number of changed event types in MobileMedia is 74% fewer than the total number of changed pointcuts.

Second, the PCD and @PCD releases showed benefit over EVT for certain design rules.

- For the EVT releases, we had to be aware of and manually maintain design rules related to encapsulating entire types (e.g. to make an entire class synchronized). The PCD, @PCD, and OM releases used pointcuts to automatically maintain such design rules.
- Such design rules show cases where patterns do not exhibit fragile pointcut behavior, as the pointcuts are expected to capture all methods in the advised types.

AOSD'12, March 25–30, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1092-5/12/03. . . $10.00

Additionally, the EVT releases showed some benefit over the @PCD releases due to its ability to uniformly access context information when announcing events.

In summary, the key contributions of the case study performed in this work are:

- The first rigorous study of different language features for four different AO interfaces on substantial case studies.
- A suite of tools to automate measuring change propagation for PCD, OM, @PCD, and EVT. This automation reduces the chance for errors in our empirical study.
- A new set of 21 MobileMedia and 30 Health Watcher releases using @PCD, OM, and EVT interfaces.
- A change propagation analysis, that shows the stability gained from designs using annotation-based pointcuts and quantified, typed events in the face of fragile pointcuts [8, 19, 22].

Next we describe some prior studies on AO interfaces. In Section 3 we introduce the studied language designs. We then present our case study in Sections 4–7. Then we conclude with discussion in Section 8 and future work.

2. Related Work

Language Feature Comparison Studies Figueiredo *et al.* [8] studied the effects of evolving software product lines (SPLs) using aspects. Similar to our study, they measure change propagation and a set of standard metrics (such as coupling and cohesion). Their study showed some of the pros and cons to using AO language features when compared to OO features. For example, their study showed that changes affecting core features (such as changing a mandatory feature into an optional feature) are not well suited for AO. However, their study was limited to only one AO interface (pattern-based pointcuts) and as such does not generalize to other AO interfaces.

Hoffman and Eugster [12] studied the coupling, cohesion and separation of concerns for several projects with implementations in Java, AspectJ, and explicit join points (EJPs). Their study focused solely on implementing exception handling with each AO interface. Similar to our study, their study examines software engineering metrics and compares each AO interface against each other. Our study however looks at a total of 4 AO interfaces and multiple types of crosscutting behavior (instead of just exception handling) in two distinct systems with a total of 68 AO releases.

Kiczales and Mezini [13] studied seven different AO interfaces for improving separation of concerns in AspectJ-like languages. These included standard method calls, explicit join points using annotation-based pointcuts and implicit pattern-based pointcuts. They analyze each mechanism based on locality, explicit/implicit and ease of evolution and then provide guidelines on when each mechanism should be used in practice. Our work is similar in the sense that we analyze several language interfaces. Their work uses a simple example for comparison while our work examines 7 releases of the MobileMedia [8] software product line and 10 releases of the Health Watcher [11, 20] web application.

Maintenance Studies Ferrari *et al.* [7] studied several SPLs to determine the possible language features that led to faults in those systems. Their results show that obliviousness was a key cause of faults in those systems and that pattern-based pointcuts are not necessarily the main cause of faults in AO designs. Their study focused on determining the cause of faults in AO systems while our study examines the effects of several AO interfaces on software maintenance.

Kulesza *et al.* [16] investigated the effect of AO interfaces on software maintenance by measuring standard software engineering metrics. They measured separation of concerns, coupling, cohesion, and size and concluded that in the presence of widely-scoped design changes, the AO designs exhibited superior stability and reusability compared to OO designs. In their study, they look at 2 releases of the Health Watcher application. Our study on the other hand examines 10 releases of Health Watcher and 7 releases of MobileMedia, giving us more variability to examine and allowing us to analyze the effects of varying types of interfaces added to a system. Their work also focuses solely on pattern-based pointcuts, whereas we consider several AO interfaces.

3. Background

In this section, we give an overview of each studied AO interface using an example based on a pattern occurring frequently in one of our case study candidates, MobileMedia.

Let us consider the class `FileScreen` shown in Figure 1. This class represents a screen presented to a user for manipulating a file. When the `saveCommand` is requested, the class saves the data to the specified file name. When the `deleteCommand` is requested, the file is deleted. The screen is shown on a `display`, which can be updated to show different screens.

An example requirement for such a class is to consistently display error messages to the user. There may be multiple screens that deal with I/O and all such screens should consistently handle errors that occur during that I/O by showing the I/O error screen. Note that in some cases, the designers have decided no error should be displayed (for example, when deleting a file and it was deleted by another user between the time of request and handling of the command).

3.1 Using Pattern-Based Pointcuts

The aspect `ExceptionHandler` shown in Figure 1 (lines 12–21) implements the requirement to consistently handle all exceptions using pattern-based pointcuts. This aspect contains an around advice (lines 15–20), which when triggered will properly handle the exception. The named pointcut `savepc` (lines 13–14) matches the execution of the method `save`, which had to be created in order to have a join point capable of being advised by the aspect. This is an example of quantification failure [24]. Note the advice uses the display variable, which is not exposed as context in the pointcut and is instead accessed indirectly through available context (the receiver object, screen).

```
1  @interface FileSaveEvent { }
2  class FileScreen {
3    Display display;

5    void handleCommand (Command c) {
6      if (c == saveCommand) { save (); }
7      else if (c == deleteCommand) { .. }
8    }
9    @FileSaveEvent
10   void save () { /* open the file and save data */ }
11 }
12 aspect ExceptionHandler {
13   pointcut savepc(FileScreen screen):
14     execution(* FileScreen.save ()) && this(screen) {
15   around(FileScreen screen): savepc(screen) {
16     try { proceed (); }
17     catch (FileNotFoundException e) {
18       screen.display.ShowFileError (e);
19     }
20   }
21 }
```

Figure 1. An example usage of pattern-based pointcuts [15]

3.2 Using Quantified, Typed Events

Quantified, typed events [19] allow programmers to declare named event types. An event type declaration p has a return type, a name, and zero or more context variable declarations. These context declarations specify the types and names of reflective information communicated between announcement of events of type p and handler methods. These declarations are independent from the modules that announce or handle these events. The event types thus provide an interface that completely decouples subjects and observers. An example event type declaration is shown in Figure 2 (line 1). The **event** FileSaveEvent declares that events of this type make one piece of context available: the display.

```
1  void event FileSaveEvent { Display display; }
2  class FileScreen {
3    Display display;

5    void handleCommand (Command c) {
6      if (c == saveCommand) {
7        announce FileSaveEvent(display) {
8          // open the file and save data
9        }
10     } else if (c == deleteCommand) { .. }
11   }
12 }
13 class ExceptionHandler {
14   void handler(FileSaveEvent next) {
15     try { next.invoke(); }
16     catch (FileNotFoundException e) {
17       next.display().ShowFileError (e);
18     }
19   }
20   when FileSaveEvent do handler;
21 }
```

Figure 2. An example usage of quantified, typed events [19]

The class FileScreen declares and announces an event of type FileSaveEvent using an announce expression (lines 7–9). Arbitrary blocks can be declared as the body of an announce expression, which avoids quantification failure. The event type FileSaveEvent declares one context variable, thus the **announce** expression binds the field display to the context variable named display (line 7).

Finally, the names of **event** declarations can be utilized for quantification in a binding declaration. A binding declaration, binding in short, associates a handler method to a set of events identified by an event type. The binding (line 20) says to run the method **handler** when events of type FileSaveEvent are announced. This allows quantifying over all announcements of FileSaveEvent with a succinct binding declaration, without depending on the modules that announce events. Use of event names in bindings simplifies them and avoids coupling observers with subjects.

Each handler method takes an event closure as the first argument. An *event closure* [19] contains code needed to run other applicable handlers and the original event's code. An event closure is run by an **invoke** expression. The **invoke** expression in the implementation of the handler method (line 15) causes other applicable handlers and the original event's code to run before handling any exceptions.

3.3 Using Annotation-based Pointcuts

When using pattern-based pointcuts, the code being advised by aspects is completely oblivious to those aspects. One approach that sacrifices some obliviousness, which is quite similar looking to quantified, typed events, is to mark each advised join point with an annotation [13]. The aspects then match based on that annotation.

For example, consider the code shown in Figure 1. The gray portions of the code represent what was changed from the pattern-based implementation. An annotation FileSaveEvent was created (line 1) and then used to mark the advised join point method save (line 10). The pointcut for the aspect (line 14) was modified to become **execution**(@FileSaveEvent * *(..)) and match based on that annotation instead of matching against the string representation of the method name.

3.4 Using Open Modules

Open modules [1] declare which join points are exposed to aspects via a module definition. This puts the burden onto the module maintainer to maintain relationships between join points in the base code and pointcuts matching those points.

Ongkingco *et al.* [17] proposed an extended version of open modules and an implementation for AspectJ [15]. We use their implementation's syntax for this example. Figure 3 is an example module for our exception handling requirement. The figure omits the class FileScreen and aspect ExceptionHandler, as they are identical to Figure 1.

The module ExHandle (lines 3–7) for the class FileScreen (Figure 1, line 4) exposes one named join

```
/* class FileScreen and aspect ExceptionHandler
   same as in Figure 1. */
module ExHandle {
  class FileScreen;
  expose :
    ExceptionHandler.savepc(FileScreen);
}
```

Figure 3. Exception handling with open modules [1, 17]

point in that class: the pointcut `savepc` defined in the aspect `ExceptionHandler` (Figure 1, line 6). The module states that the aspect is allowed to match this join point. If the signature of the pointcut changes in the aspect, the maintainer of the module would be required to update the module definition as well.

4. Case Study Overview

To evaluate the proposed AO interfaces studied here, we examined them in the context of two applications: an existing software product-line application called MobileMedia [8] and an existing web application called Health Watcher [11, 20]. Thus, both cases are already vetted.

This section describes our experimental setup, the technique used to generate new releases of the studied applications and the tools developed and used for the study.

4.1 Experimental Setup

In order to perform this case study, we created a total of 51 modified releases of the MobileMedia and Health Watcher applications, modified 2 compilers to automatically compute various software engineering metrics and created a tool for automatically measuring change propagation. All artifacts and tools are available for download[1]. An important advantage of these tools was that they removed the manual, and often error-prone, steps from our empirical study. In this section, we describe each tool in detail.

4.1.1 New Code Artifacts

The OO and pattern-based pointcut code artifacts for this study were re-used from previous work [8, 11, 20]. Since the artifacts using annotation-based pointcuts, open modules and quantified, typed events did not previously exist for either application, we created them. When creating these artifacts, *our objective was to keep other variables such as design strategy constant between all versions and only change the crosscutting feature*.

For example, starting with release 4 of the Health Watcher releases for AspectJ, an observer pattern aspect library was used. This library was re-used in the annotation-based pointcut, open modules and Ptolemy releases despite the fact that Ptolemy's quantified, typed events actually make this library unnecessary (the events implement the observer pattern directly, so no library is needed). Removing this library however would change the base and aspect components in the system and introduce extra variables into the analysis. Leaving it in place meant the only difference between the Ptolemy and other releases was the quantification mechanism used for implementing crosscutting behavior.

Creating Annotation-based Pointcut Releases Using the pattern-based pointcut releases as a starting point, we implemented all 7 releases of MobileMedia and all 10 releases of Health Watcher using an annotation-based pointcut syntax [13]. We modified each pointcut in every aspect to match based on a new annotation and for each join point in the base code which matched the original pointcut, we annotated the method with the new annotation. The names of the annotations were chosen based on properties of the code, following the guidelines of Kiczales and Mezini [13]. The results were verified by comparing the weaving logs produced by the standard AspectJ compiler (ajc) for both the original pattern-based pointcut releases and the new annotation-based pointcut releases.

Creating Open Module Releases To study the effect of open modules [1], we implemented all 7 MobileMedia and all 10 Health Watcher releases using the AspectJ-based implementation of open modules [17]. Starting with the first release, we made a copy of the pattern-based pointcut release and then created module definitions.

For each subsequent release, we copied the pattern-based pointcut release and then copied and updated the module definition(s) from the previous open modules release. Modules were updated to reflect changes in the base code and, where appropriate, new modules were added. Modules were created to follow the package structure of the system, following the recommendation of Ongkingco *et al.* [17].

Creating Quantified, Typed Event Releases For each quantified, typed event [19] release we started with the pattern-based pointcut release as a template, creating one handler class for each aspect. For each advice body in an aspect, a new handler method was added to the handler class. Event types were created and event announcement added to emulate the pattern-based pointcut-advice semantics.

Note that since the initial work on Ptolemy [19], the language has been extended to include support for inter-type declarations. The syntax is identical to that of AspectJ and the implementation was directly borrowed from the ABC AspectJ compiler [2] and added to the Ptolemy compiler, as the research version of the Ptolemy compiler is also based on the JastAdd [5] extensible compiler framework.

4.1.2 Automation of Empirical Evaluation

Evaluating the benefits of the studied designs using standard software engineering metrics and change propagation by hand can be tedious and error-prone. To solve this problem, we built several tools to automatically measure these metrics and allow for consistency. These included several modified

[1] **Tools/artifacts download:**
http://ptolemy.cs.iastate.edu/design-study/

compilers and tools for measuring change propagation. Our tool support builds on the open-source ABC [2] AspectJ compiler. The ABC compiler was used for two reasons: it has a JastAdd [5] extensible frontend available which simplifies extensions and it contains support for the only known implementation of open modules. The use of ABC was also driven by the fact that the research version of the Ptolemy compiler is also JastAdd-based and our tool extensions could be re-used for both compilers (giving us automated tool support for every studied language design).

Measuring Change Propagation To measure change propagation, a JastAdd module was created to serialize the parsed AST into an XML format. Since every compiler used in our study is based on the JastAdd extensible compiler, the new functionality was shared as a reusable module between these compilers. This also ensured that change propagation measurement was done consistently.

A separate tool was created that takes two of these XML files as input, representing two versions of the same code tree, and compares the two trees to determine which components are new, were removed, or have changed. We considered a renamed component (including moving it to another package) as a change (instead of a remove and an add) and manually identified such renames in a separate XML file to aid the tool.

The tool is capable of determining changes at the granularity of classes, aspects, event types, annotations, and pointcuts. The results were then manually verified against diffs of the MobileMedia code releases for Java and AspectJ to ensure the accuracy of the tool.

Measuring Software Engineering Metrics To measure the metrics suite proposed by Chidamber and Kemerer [4] for coupling and cohesion, we created another JastAdd module which measures and reports these metrics. This module was shared and used in each compiler in our study.

Chidamber and Kemerer propose that a component is coupled to another component if it accesses a field or calls a method from the other component. They also propose a class is cohesive if the operations of the class operate on similar attributes of the class.

We used the previous results from Figueiredo *et al.* [8] as a guide for our implementation, comparing the values for the OO and pattern-based pointcut MobileMedia releases to their previously published results. No extension was necessary for the annotation-based pointcuts, as the existing OO and pattern-based pointcut metrics apply directly. The metrics suite was extended to support open modules and quantified, typed events in a straight-forward manner, similar to the pattern-based pointcut extensions.

For open modules releases, modules are treated like classes and join point exposures treated similar to an aspect pointcut. For quantified, typed events, event type declarations are treated like a class with context access considered a field of the event type. Announcing an event is treated like a method call.

4.2 Threats to Validity

In this section we discuss internal and external threats to the validity of our case study.

4.2.1 Internal Validity

To reduce the risk of bias when selecting languages for study, we first decided the focus of the study to be examining the effect of AO interfaces for minimizing pointcut fragility and change propagation. Then we categorized existing AO interfaces by how they achieve quantification.

In the first category, quantification is controlled solely by aspects and pattern-based pointcuts was the most relevant choice in this category as it is used in industry and also highly researched. In the second category, quantification is controlled solely by the base code. In this category, there were two candidates: aspect-aware interfaces [14] and open modules [1]. We picked open modules because an implementation was available for it. In the third category, quantification is controlled by an intermediary between base components and aspects. There were several candidates: XPIs [24], annotation-based pointcuts, implicit invocation with implicit announcements [21] and quantified, typed events [19]. We picked quantified, typed events due to our familiarity with its compiler infrastructure and also picked annotation-based pointcuts as the language was not developed by the authors and compiler support was readily available.

The code artifacts created for this study were the MobileMedia and Health Watcher (releases for annotation-based pointcuts (@PCD), open modules (OM), and quantified, typed events (EVT). To reduce the risks associated with creating these artifacts, we attempted to keep other variables constant (such as design strategy used) and only vary the quantification mechanism used.

We reduced the risk associated with creating the EVT, @PCD, and OM releases by first basing them off the existing pattern-based pointcut releases (which were not created by any of the authors). Next, we used recommendations by experts in each respective language in their published work [13, 17, 19] to modify the pattern-based pointcut releases and create the releases for the new AO interfaces.

For example, we followed the guidelines given by the implementers of the open modules implementation used to create one module definition for each package [17]. We also followed a naming scheme proposed by Kiczales and Mezini [13] when generating annotations for the @PCD releases, which was shown to offer design stability.

4.2.2 External Validity

The main concern regarding external validity that we identified is the studied systems may not faithfully represent software in industry. This risk is reduced since the applications are implemented in both Java and AspectJ, which is a representative approach in the AO domain. Further, MobileMedia is a software-product line comprised of 8 releases based on industry-strength technologies for mobile systems, such as the Java Mobile Information Device Profile (MIDP) and Mobile Media API (MMAPI). Additionally, this system has been studied extensively [7–9, 11].

Similarly, Health Watcher is a real-world application used for reporting health complaints. This system uses several industrial strength technologies/techniques, such as persistence mechanisms, remote invocation (RMI), concurrency, JDBC, etc.

5. Case Study: MobileMedia

This section contains our first studied project, a software product-line application called MobileMedia [8]. MobileMedia is an extension of MobilePhoto [25], which was developed to study the effect of AO designs on software product lines (SPL). MobileMedia is an SPL for applications that manipulate photos, music, and videos on mobile devices. MobileMedia extends MobilePhoto to add new mandatory, optional and alternative features.

5.1 Change Propagation Analysis

A key benefit of a modular software design is in its ability to hide design decisions that are likely to change [18]. Thus, we consider the number of changed components as a result of a changed design decision to be an important comparator for a software design. To quantify this, similar to Figueiredo *et al.* [8], we measured the number of added, removed, and changed components in each system for each release.

5.1.1 Component Changes

The changes to base components are shown in Figure 4. This table includes the pure Java releases (OO), pattern-based pointcut releases (PCD), annotation-based pointcut releases (@PCD), open modules releases (OM), and the quantified, typed event releases (EVT). This table considers Java classes and interfaces, aspects, and open modules.

Note that the declarations of annotations and event types are not included in the counts for this table, as they are measured separately and considered in the next section to give a direct comparison to pointcuts.

Components Added For all releases, new components added in the pattern-based pointcut (PCD) releases were also added to the annotation-based pointcut (@PCD), open modules (OM), and quantified, typed event (EVT) releases. Note that the @PCD values are identical to the PCD values.

In R2, R6, R7, and R8, the number of added components differs for the open modules (OM) releases compared to PCD (marked in bold) due to the addition of modules in each of those releases. All aspects and base components in the OM releases are identical to the PCD releases.

In R4, the releases with pointcuts (PCD, @PCD, and OM) added an aspect that only handles precedence. This aspect was not added in the quantified, typed event release, as precedence in that release is controlled by the order of registering handler classes. This registration occurs inside the main class.

			R2	R3	R4	R5	R6	R7	R8	Total
Components	Added	OO	**9**	**1**	**0**	**5**	**7**	**10**	**6**	38
		OM	**17**	2	3	6	**11**	**17**	**22**	78
		PCD	13	2	3	6	8	14	16	62
		@PCD	13	2	3	6	8	14	16	62
		EVT	13	2	**2**	6	8	14	16	61
		Differences to PCD marked in BOLD blue								
	Removed	OO	**0**	0	0	0	0	1	**1**	2
		OM	1	0	0	0	0	1	0	2
		PCD	1	0	0	0	0	1	0	2
		@PCD	1	0	0	0	0	1	0	2
		EVT	1	0	0	0	0	1	0	2
	Changed	OO	5	**8**	**5**	**8**	**6**	**19**	**17**	68
		OM	5	**14**	**6**	**13**	**6**	**34**	**26**	104
		PCD	5	10	2	10	5	27	18	77
		@PCD	5	**8**	2	**11**	**7**	27	**20**	80
		EVT	5	**9**	**1**	**8**	5	**25**	**20**	73

Figure 4. Base components change propagation in MobileMedia for each release

Components Removed In all 7 changed releases (R2–R8), the AO releases all have the same components removed. In R2, the PCD release removed a class `BaseThread` and in R8 the OO release removed the class `SplashScreen`. Since we did not implement either of the OO or PCD releases, we simply mimicked these changes in the @PCD, OM, and EVT releases.

Components Changed The difference between the components changed for the pattern-based pointcuts (PCD) and open modules (OM) releases is due entirely to changes in the modules, as once again all aspects and base components in the OM releases are identical to the PCD releases. Starting with R3, each release modified modules from the prior release due to changes in the aspects.

In R3, the PCD release changes two more components (`UtilAspectEH` and `ControllerAspectEH`) than the @PCD release due to the fragility of the pointcuts in those components. However, in R5, R6, and R8 the @PCD releases change more base components than the PCD releases, despite avoiding the fragile pointcut problem with existing pointcuts. This is due to the need to annotate the base code with new annotations.

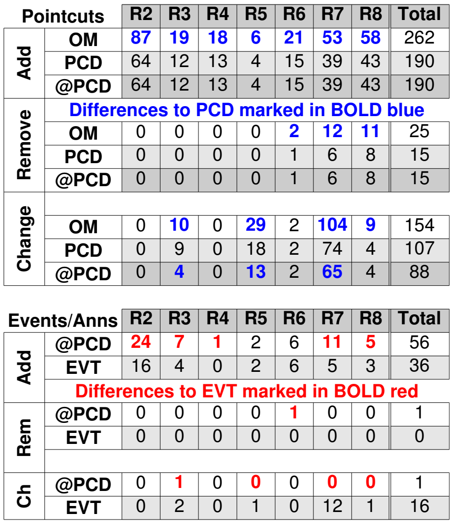
The difference in changes for R3 between @PCD and EVT was due to a changed event type requiring a change in the signature of the handler method.

For R4 however, the difference in values represents two important differences in the AO interfaces. First, the changed component in EVT was due to adding a precedence declaration to a handler (in @PCD this was a new aspect, not a changed aspect). Second, the two changed components in PCD and @PCD were from refactoring base code to expose join points. EVT did not need to perform such refactorings as it allows arbitrary statements as event announcements.

Of the remaining 7 changes that occurred in @PCD and not EVT, 3 were due to updating the precedence aspect, 1 was due to exposing join points and the remaining 3 were from changes in context (which for EVT shows up as changes in the event types).

5.1.2 Quantification Mechanism Changes

The change propagation results are shown in Figure 5. The table lists the number of pointcuts added, changed, or removed for the open modules (OM), annotation-based pointcut (@PCD), and pattern-based pointcut (PCD) releases. The number of annotations added, changed or removed are shown for the @PCD releases and the number of event types added, changed, or removed are shown for the EVT releases.



	Pointcuts	R2	R3	R4	R5	R6	R7	R8	Total
Add	OM	**87**	**19**	**18**	**6**	**21**	**53**	**58**	262
	PCD	64	12	13	4	15	39	43	190
	@PCD	64	12	13	4	15	39	43	190
Remove	**Differences to PCD marked in BOLD blue**								
	OM	0	0	0	0	**2**	**12**	**11**	25
	PCD	0	0	0	0	1	6	8	15
	@PCD	0	0	0	0	1	6	8	15
Change	OM	0	**10**	0	**29**	2	**104**	**9**	154
	PCD	0	9	0	18	2	74	4	107
	@PCD	0	**4**	0	**13**	2	**65**	4	88

	Events/Anns	R2	R3	R4	R5	R6	R7	R8	Total
Add	@PCD	**24**	**7**	**1**	2	6	**11**	**5**	56
	EVT	16	4	0	2	6	5	3	36
Rem	**Differences to EVT marked in BOLD red**								
	@PCD	0	0	0	0	**1**	0	0	1
	EVT	0	0	0	0	0	0	0	0
Ch	@PCD	0	**1**	0	**0**	0	**0**	**0**	1
	EVT	0	2	0	1	0	12	1	16

Figure 5. AO interfaces change propagation in MobileMedia for each release

Pointcuts The pointcuts added, removed, and changed were measured for all releases with pointcuts (PCD, @PCD, and OM) and there are two sets of comparisons to note. First, the OM releases have more pointcuts added and changed in almost every release (marked in bold) when compared to the PCD releases. This is due to the additional pointcuts contained in the module definitions.

The second comparison is between the PCD and @PCD releases. In three releases, the @PCD releases have fewer changed pointcuts. This occurred due to the gained stability from using the annotation-matching pointcut syntax. In total, the @PCD releases have almost 18% fewer changed pointcuts compared to the PCD releases.

Annotations and Events The annotations for the @PCD releases and the event types for the EVT releases are similar in that both mark join points in the base code for aspect code to advise. The change propagation of these two mechanisms is also similar. The differences between them (marked in bold) occur for several reasons.

The event types in EVT contain typed context declarations, while annotations do not contain any context. As such, when types in the base code (used as context) change, any event type referencing those types must also be updated. This is why the annotations have no changes in any @PCD release (the change in R3 was a renamed annotation) and the EVT releases have several changes.

In R2, the difference in the number of added event types and annotations is due to EVT's lack of quantification failure. For example, the @PCD release had to create an annotation to mark a join point for use in a `within` pointcut due to quantification failure. The EVT release was also able to re-use more event types than the @PCD release, saving the addition of 7 event types.

Pointcuts vs Annotations/Events In R7 a mandatory feature was turned into two alternative features, leading to changes in the base components which propagated to the event types and event handlers for the EVT release. 10 of the 12 resulting event type changes were due to the renaming of base components passed as context in those events types. Consider on the other hand the PCD release which required changing 38 of the 74 pointcuts due to the fragility of those pointcuts.

In R8, several new alternate features were added to the system. The EVT release was able to re-use several existing event types, leading to the addition of only 3 new event types. Similarly, the @PCD release only required the addition of 5 new annotations. The PCD release however required adding 43 new pointcuts to the system.

In general, note that the total number of added event types and annotations are 81% and 70% fewer, respectively, than the total number of added pointcuts for PCD releases. Also note that the total number of changed event types is 85% fewer than the total number of changed pointcuts in the PCD releases and 82% fewer than the total pointcuts changed in the @PCD releases.

5.1.3 Summary

In summary, for some releases quantified, typed events showed an improved ability to withstand changes in components. In particular, for releases where significant refactoring in the base components took place, the EVT designs were able to reduce the impact of these changes in the base code from the handlers. Additionally,

- the total number of added event types and annotations are less than a third the number of pointcuts added in the PCD releases, showing that event types and annotations are re-used by multiple pointcuts,
- the total number of changed event types is 85% fewer than the total number of changed pointcuts in the PCD releases and 82% fewer than the total pointcuts changed in the @PCD releases,

- the @PCD releases have almost 18% fewer changed pointcuts compared to the PCD releases due to the lack of fragile pointcuts, and
- the EVT and @PCD releases were both able to efficiently re-use events/annotations leading to fewer additions in releases adding alternate features.

5.2 Software Engineering Metrics

As previously discussed, the main difference between most AO interfaces and quantified, typed events is that the dependency between components that announce events is explicitly stated using announce expressions that name event types. With most AO interfaces, this dependency is implicitly defined by the language semantics. Explicitly naming event types or annotations introduces coupling. The main goal of this section is to study the change in coupling between components. In order to perform this evaluation, we used a subset of the metrics suite proposed by Chidamber and Kemerer [4], Fenton and Pfleeger [6], and subsequently refined by Garcia *et al.* [8, 10].

		R2	R3	R4	R5	R6	R7	R8
CBC	OO	32	40	40	65	80	103	131
	OM	35	50	59	94	121	159	217
	PCD	35	50	59	94	121	159	217
	@PCD	82	106	122	161	200	255	332
	EVT	74	100	120	159	203	271	371
LCOO	OO	123	194	224	241	296	311	365
	OM	147	244	266	259	369	502	534
	PCD	147	244	266	259	369	502	534
	@PCD	147	244	266	259	369	502	534
	EVT	123	162	171	257	365	426	539

Figure 6. Coupling and Cohesion for MobileMedia

Coupling Coupling between components (CBC) [4] is a measurement of coupling. A component is coupled to another component if it accesses a field or calls a method on it. Figure 6 shows the results of our measurements.

The @PCD and EVT releases all have upwards of twice as much explicit coupling in the system compared to the PCD and OM releases. This is due to the explicit marking of join points (with annotations and event type announcements). However, realize that the added coupling is not coupling between aspects and base code but rather aspects to event types and base code to event types. Thus, this coupling only creates a maintenance issue if an event type changes (such as in R7).

Cohesion Lack of cohesion in operations [4] (LCOO) is a measurement of cohesion of the classes in the system, based on how similar operations use attributes of the class. If methods of a class operate on the same attributes, the class is said to be cohesive and has a lower LCOO value. LCOO for all releases was measured and is shown in Figure 6. Note that the PCD, @PCD, and OM releases all have the same values due to having the same methods/fields in classes and ITDs/advice in aspects.

In general, quantified, typed events have more cohesion (indicated by lower LCOO) than the pointcut-based approaches. This is mostly due to the lack of needing to refactor the base code to expose join points to the aspect code. Such refactored code often only works on a small sub-set of the fields in the class, making the class less cohesive.

		R2	R3	R4	R5	R6	R7	R8
LOC	OO	1159	1314	1363	1555	2051	2523	3016
	OM	1337	1570	1700	1928	2474	3207	3999
	PCD	1276	1494	1613	1834	2364	3068	3806
	@PCD	1452	1723	1852	2094	2664	3461	4257
	EVT	1427	1669	1781	2050	2646	3398	4254
NOC	OO	24	25	25	30	37	46	51
	OM	31	33	36	42	53	69	91
	PCD	27	29	32	38	46	59	75
	@PCD	51	60	64	72	85	109	130
	EVT	47	53	56	64	78	96	115
NOA	OO	62	71	74	75	106	132	165
	OM	82	99	108	112	149	187	237
	PCD	62	72	76	77	110	139	177
	@PCD	62	72	76	77	110	139	177
	EVT	71	92	96	101	144	175	217
NOO	OO	124	140	143	160	200	239	271
	OM	143	169	179	197	247	308	369
	PCD	143	169	179	197	247	308	369
	@PCD	143	169	179	197	247	308	369
	EVT	142	167	177	196	245	302	378

Figure 7. The measured size metrics for MobileMedia

Size Metrics Figure 7 shows the number of components (NOC) and total lines of code (LOC) for each release. The number of components includes classes and interfaces for all releases. For the PCD, @PCD, and OM releases it also includes aspects. For OM it includes modules, @PCD includes annotations and EVT includes event types.

Lines of code were measured using a tool[2] that ignores comment and whitespace lines. All other lines were included and every component from NOC was included.

Number of operations (NOO) was measured as the total number of methods in classes, introduced methods in aspects, advice bodies in aspects and handler methods in event handlers. Number of attributes (NOA) was measured as the total number of fields in classes or aspects (including intertype declared fields) and the number of context variables in quantified, typed events.

As one would expect from creating so many events and annotations, the lines of code and number of components is higher for both @PCD and EVT. The number of attributes is also higher for EVT due to counting event type context variables as attributes.

Summary In summary, our results show the total coupling is higher in the annotation-based pointcut and quantified, typed event releases due to the interface added between base components and aspects. The increased coupling is a tradeoff for the stability gained by the interface between aspect and base code, as the previous section clearly demonstrates.

[2] Retrieved from: `http://reasoning.com/downloads.html`

6. Case Study: Health Watcher

This section contains our second studied project, a web-based application called Health Watcher [11, 16, 20]. Health Watcher is an application for users to file health complaints. The system was initially developed in 2001 and has undergone 9 releases to add new features and fix previous bugs.

6.1 Change Propagation Analysis

As stated in the previous case study, we consider the number of changed components as a result of a change in a design decision to be an important comparator for a software design. This section performs our analysis on Health Watcher.

6.1.1 Component Changes

The changes to base components are shown in Figure 8. This table includes the Java releases (OO), pattern-based pointcut releases (PCD), annotation-based pointcut releases (@PCD), open modules releases (OM), and the quantified, typed event releases (EVT). This table considers Java classes/interfaces, aspects, and open modules.

			R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total
Components	Added	OO	**88**	**4**	**12**	**2**	**3**	4	4	**4**	12	**5**	138
		OM	**106**	**12**	16	**4**	0	4	4	2	12	6	166
		PCD	101	11	16	3	0	4	4	2	12	6	159
		@PCD	101	11	16	3	0	4	4	2	12	6	159
		EVT	**100**	**10**	16	3	0	4	4	2	12	6	157
	Differences to PCD marked in BOLD blue												
	Removed	OO	0	0	0	0	**1**	0	0	0	0	**2**	3
		OM	0	0	0	0	0	0	0	0	0	1	1
		PCD	0	0	0	0	0	0	0	0	0	1	1
		@PCD	0	0	0	0	0	0	0	0	0	1	1
		EVT	0	0	0	0	0	0	0	0	0	1	1
	Changed	OO	0	**22**	**6**	**15**	**16**	2	27	3	**23**	**48**	162
		OM	0	**27**	**9**	**9**	1	**3**	27	**5**	**23**	**55**	159
		PCD	0	25	8	7	1	2	27	3	22	52	147
		@PCD	0	**26**	8	**29**	1	2	27	3	**23**	**55**	174
		EVT	0	**26**	8	**32**	1	2	27	3	**23**	**54**	176

Figure 8. Base components change propagation in Health Watcher for each release

Components Added For all releases, new components added in the pattern-based pointcut (PCD) releases were also added to the annotation-based pointcut (@PCD), open modules (OM), and quantified, typed event (EVT) releases. Note that the @PCD values are identical to the PCD values (as annotations are considered separately in Figure 9).

In R1, R2, and R4, the number of added components differs for the open modules (OM) releases compared to PCD (marked in bold) due to the addition of modules in each of those releases. All aspects and base components in the OM releases are identical to the PCD releases.

In R1, an aspect that only contains a declare parents statement was not added in the EVT release. This statement failed to compile with the abc based intertype declarations implementation. Instead, we manually modified the base classes to add the Serializable interface to the 2 types. This particular aspect did not change in the PCD releases, thus our work-around did not cause problems in later EVT releases.

In R2, the releases with pointcuts (PCD, @PCD, and OM) added an aspect that only handles precedence. This aspect was not added in the quantified, typed event release, as precedence in that release is controlled by the order of registering handler classes. This registration occurs inside the main class or using annotations in the handler classes.

Components Removed Similar to MobileMedia, in all 9 changed Health Watcher releases (R2–R10), the AO releases all have the same components removed.

Components Changed Unlike MobileMedia where the difference between the components changed for the pattern-based pointcuts (PCD) and open modules (OM) releases was due entirely to changes in the modules, in Health Watcher some of the aspects also were modified in order to give anonymous pointcuts names (for the modules to reference).

Also unlike MobileMedia, the components changed for @PCD and EVT are more in Health Watcher for R4 than the PCD release due to needing to add annotations and event announcements in base code. This was because fewer base components changed in the PCD release but over 20 had to be modified to add annotations and event announcement.

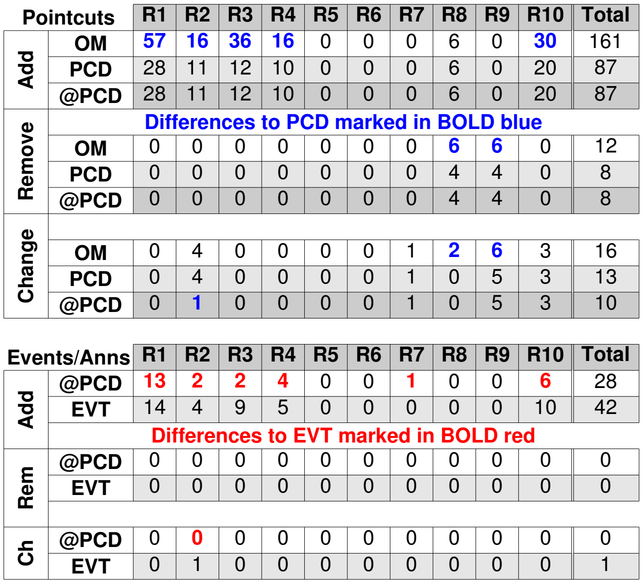
6.1.2 Quantification Mechanism Changes

The change propagation results in terms of modularization techniques are shown in Figure 9. The table lists the number of pointcuts added, changed, or removed for the open modules (OM), annotation-based pointcut (@PCD), and pattern-based pointcut (PCD) releases. The number of annotations added, changed or removed are shown for the @PCD releases. It also lists the number of event types added, changed, or removed for EVT.

Pointcuts Again, the pointcuts added, removed, and changed were measured for all releases with pointcuts (PCD, @PCD, and OM). Once again, the OM releases have more pointcuts added and changed compared to the PCD releases, as the module definitions also contain named pointcuts.

Unlike the MobileMedia case study, Health Watcher had relatively stable pointcuts. As such, the only benefit observed in the @PCD releases occurred in R2, where 3 fewer pointcuts were changed.

Annotations and Events Unlike the MobileMedia case study, annotations and event types in Health Watcher perform roughly the same in all releases, with the exception of R3. In this release, there were several (4) events that had to be duplicated: once with a void return type and once with a non-void return type. This was due to the advice being applied to multiple methods (with differing return types). The PCD releases simply marked all methods with the same annotation and the aspect was able to advise them all, without regard to the return type. This problem also accounts for the extra events in R1 and R2.



Pointcuts		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total
Add	OM	**57**	**16**	**36**	**16**	0	0	0	6	0	**30**	161
	PCD	28	11	12	10	0	0	0	6	0	20	87
	@PCD	28	11	12	10	0	0	0	6	0	20	87
Remove	Differences to PCD marked in BOLD blue											
	OM	0	0	0	0	0	0	0	**6**	**6**	0	12
	PCD	0	0	0	0	0	0	0	4	4	0	8
	@PCD	0	0	0	0	0	0	0	4	4	0	8
Change												
	OM	0	4	0	0	0	0	1	**2**	**6**	3	16
	PCD	0	4	0	0	0	0	1	0	5	3	13
	@PCD	0	**1**	0	0	0	0	1	0	5	3	10

Events/Anns		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total
Add	@PCD	**13**	**2**	**2**	**4**	0	0	**1**	0	0	**6**	28
	EVT	14	4	9	5	0	0	0	0	0	10	42
	Differences to EVT marked in BOLD red											
Rem	@PCD	0	0	0	0	0	0	0	0	0	0	0
	EVT	0	0	0	0	0	0	0	0	0	0	0
Ch	@PCD	0	**0**	0	0	0	0	0	0	0	0	0
	EVT	0	1	0	0	0	0	0	0	0	0	1

Figure 9. AO interfaces change propagation in Health Watcher for each release

Pointcuts vs Annotations/Events In general, note that the total number of added event types and annotations are 52% and 68% fewer, respectively, than the total number of added pointcuts for PCD releases. This result is similar to the MobileMedia results.

6.1.3 Summary

In summary, the Health Watcher case study showed similar results to the MobileMedia case study. The noticeable differences between the studies were due to the fact that the Health Watcher study tended to simply add new aspects and avoid changing existing aspects and base code as much as possible while the MobileMedia study made significant modifications (in order to change mandatory features into optional ones).

6.2 Software Engineering Metrics

The software engineering metrics for Health Watcher followed the same trends as for Mobile Media and thus were omitted for space reasons.

7. Key Observations

We observed several key benefits to the studied designs. These benefits are described in detail in this section.

7.1 Inter-Type Declarations

A static feature of AspectJ that allows adding fields/methods to other classes is inter-type declarations (ITDs) [15]. This feature was recently added to the Ptolemy language (with the same syntax as AspectJ) and thus available for all studied AO interfaces.

In MobileMedia, ITDs are used mostly for two purposes: to add additional data (fields) to existing types (and manipulate that new data) and to provide alternate implementations of features. ITDs first show up in R3 and are heavily used in later releases which contain alternate features. For example in R8, ITDs are defined in 12 out of the 22 aspects (54%).

In Health Watcher, ITDs are used to add methods for timestamping complaints, starting the remote server for RMI and to implement a singleton pattern. In the system, a total of 3 out of 26 aspects (11.5%) contain ITDs.

7.2 Declare Parents

Similar to ITDs, type hierarchies in the base components can be extended in a modular manner using AspectJ's *declare parents*. This feature seems well suited to help handle alternate features in a system, but was not heavily used by the current design of the MobileMedia product-line.

In MobileMedia, only R8 contains declare parents statements to extend two type hierarchies by adding a new superclass to the base components. For the PCD, @PCD, and OM releases these effects were modular. For the EVT releases, the base components had to be modified (due to a compiler bug) and these changes were non-modular, but not invasive.

In Health Watcher, declare parents statements appear in 6 out of 26 (23%) aspects. The statements are used in several places to place marker interfaces onto a set of types, which are then advised by the pointcut patterns. This was a useful pattern in this system and while Ptolemy supports declare parents statements, the lack of a pattern form of quantification meant that these marker interfaces were not useful in those releases.

7.3 Quantification Support

Quantified, typed events give the programmer the ability to add event announcement for any arbitrary statement in the base components. The pattern and annotation-based pointcut approaches can only advise join points available in the provided pointcut language, such as method executions or calls. This often results in what Sullivan *et al.* called quantification failure [24] and is caused by incompleteness in the language's event model. Quantification failure occurs when the event model does not implicitly announce some kinds of events and hence does not provide pointcut definitions that select such events [24].

In MobileMedia, we observed several instances of quantification failure. For example, in R2 the aspects needed to advise a *while* loop and similarly in R3 the aspects needed to advise a *for* loop. To accommodate this, all pointcut-based releases (PCD, @PCD, OM) refactor the base components, for example moving these loops into newly added methods. By R8, a total of 5 refactorings were made to expose join points. This accounts for approximately 5% of the advised join points. The EVT releases did not suffer from this problem and thus these refactorings were not necessary.

In Health Watcher, we observed a different form of quantification failure. However, this time the failure was in the EVT releases and related to the handling of design rules that encapsulate entire types. As previously mentioned, the PCD releases used declare parents statements to add marker interfaces to several types. The aspects then used pattern pointcuts to target all method executions in sub-types of that marker interface. This was used for things such as making all methods in a class synchronized. Figure 10 shows the implementation for this design rule in PCD, which uses the marker interface `SynchronizedClasses` on two types and around advice to wrap the execution of all methods in those types in a **`synchronized`** statement.

```
private interface SynchronizedClasses {};
declare parents: EmployeeRepositoryArray ||
  ComplaintRepositoryArray implements SynchronizedClasses;
Object around(Object o): this(o) &&
    execution(* SynchronizedClasses+.*(..)) {
  synchronized(o) { return proceed(o); }
}
```

Figure 10. Pattern-based pointcut version of a design rule to encapsulate 2 types and make all their methods synchronized

For the EVT releases, we had to manually track this design rule across the releases. This meant that if the types involved added new methods we would need to remember the design rule and ensure those new methods also announced the proper event. For the Health Watcher example, this maintenance scenario did not occur (as the types involved did not evolve across releases) but it is important to note that we still had to be aware of the design rules and check them in each release - something the PCD releases did not require.

7.4 Fragile Pointcut Problem

As mentioned by Figueiredo *et al.* [8], the pattern-based pointcut releases of MobileMedia suffer from a fragile pointcut problem [8, 19, 22]. This could be observed in R7, where a mandatory feature PHOTO is generalized into two alternative features PHOTO or MUSIC. This required modifying many pointcuts previously relying on an implicit matching of signatures in the base components.

The renaming of the base components itself is not a problem in the EVT releases and in fact requires no modification of events or handlers; the handlers will match on the event type which remains unchanged. If the event type is renamed (for example, to remain consistently named to the base components) then all handlers and events for that event type must be updated accordingly. The key difference in these two scenarios is that in the PCD case, the developer must be aware of which pointcuts matched the given join point (which can be aided with tools such as AJDT) while in the EVT case, the compiler will specify type errors for every publisher and subscriber for that event type, eliminating fragile pointcuts.

Since @PCD releases are structurally similar to the EVT releases, they also benefited from a lack of fragile pointcuts. Similarly, the OM releases also benefited from a lack of fragile pointcuts.

Fragile pointcuts were observed in releases 3, 5, and 7. In total, 19 out of the 107 pointcuts changed (18%) across all releases were due to fragile pointcuts. This problem has already been demonstrated in small examples, however, its appearance in PCD releases of MobileMedia presents real evidence that it could affect maintenance of PCD systems. The ability of EVT, @PCD, and OM to mitigate these risks shows that such problems, when they occur in practice, can be solved using these different AO interfaces.

7.5 Access to Context Information

AspectJ provides means to access context information from advised join points [15]. The type of information available to advice however is limited by the language, such as the receiver object, method arguments, etc. In MobileMedia and Health Watcher, there were several instances where this lack of flexible availability to context added complexity to the system. For example, the exception handling aspects needed access to a field in the controller class being advised. Thus, the field needed marked public, a getter method added, or the aspect marked as privileged. Either way, the aspect becomes coupled to the interface of the advised class. This was a problem for all pointcut-based releases (PCD, @PCD, OM).

This was also a key difference between the @PCD and EVT releases. While the @PCD releases provided similar benefits in terms of preventing fragile pointcuts, in terms of context exposure the annotations were not a sufficiently expressive quantification mechanism when compared to EVT.

8. Discussion

It is important to note that our measurements of the open module releases are all based on the AspectJ-based implementation of open modules [17] (which to our knowledge is the only open modules implementation available). Thus, some of the measured differences we see are due not necessarily to open modules as an AO interface but instead due to this specific implementation.

The use of named pointcuts in this implementation required copying the full pointcut signature to the module definition. This has the effect of requiring updating two locations (the original pointcut definition and the module) if that signature changes. The pointcut signatures change any time the exposed context types changed. Any difference in the number of changed pointcuts between the PCD and OM releases of Figure 5 and Figure 9 are a result of this problem.

Specifically, for MobileMedia this was a large problem in two releases (5 and 7) as a number of base components were renamed. This renaming caused multiple pointcuts to change and that effect was duplicated in the module definitions. This problem does not necessarily manifest itself in Open Modules, as originally defined by Aldrich but is an artifact of this specific implementation.

9. Conclusion and Future Work

Finding a good separation of concerns is an important problem. It is vital for improving the reliability and evolution of software systems. New modularization techniques enable improved separation of concerns. Their invention and refinement is thus equally important for maintaining intellectual control on the growing complexity of software systems. Pattern-based [15] and annotation-based [13] pointcuts, open modules [1, 17], and quantified, typed events [19] are examples of such modularization mechanisms.

In this paper, we presented a rigorous evaluation of these AO interfaces on two already well-substantiated case studies [8, 20]. The results of our change propagation and analysis using standard design metrics [4, 6, 10] show that annotation-based pointcuts and quantified, typed events help limit the impact of change, at the cost of increased explicit coupling. This coupling however is generally not a problem as it is to interface-like entities (annotations and event types), not between base components and/or aspects.

Despite the similarites, quantified, typed events have several benefits over annotation-based pointcuts. Event types are flexible and do not suffer from quantification failure. Additionally, the uniform access to context information avoided the need to break encapsulation by exposing fields to make them available to the aspects.

In the future we plan to perform a net options value analysis [3, 23] to investigate the trade-off between the higher coupling of the annotation-based pointcut and quantified, typed event releases and the stability provided by their interfaces between aspect and base code.

Acknowledgments This work was supported in part by the NSF grant CCF-10-17334. Mehdi Bagherzadeh and Youssef Hanna provided useful comments and discussion.

References

[1] J. Aldrich. Open Modules: Modular reasoning about advice. In *ECOOP '05*, pages 144–168, 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD*, pages 87–98, 2005.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE TSE*, 20(6):476–493, 1994.

[5] T. Ekman and G. Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69 (1-3):14–26, 2007.

[6] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 1998.

[7] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *ICSE'10*, pages 65–74, 2010.

[8] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, 2008.

[9] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In *FSE*, 2006.

[10] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD*, pages 3–14, 2005.

[11] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dósea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP*, pages 176–200, 2007.

[12] K. J. Hoffman and P. Eugster. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *30th International Conference on Software Engineering (ICSE)*, pages 91–100, 2008.

[13] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP '05*, pages 195–213, 2005.

[14] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *27th international conference on Software engineering (ICSE)*, pages 49–58, 2005.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[16] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *International Conference on Software Maintenance (ICSM)*, pages 223–233, 2006.

[17] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *AOSD '06*, pages 39–50, 2006.

[18] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12): 1053–8, December 1972.

[19] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, 2008.

[20] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *17th conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 174–190, 2002.

[21] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM '10*, 20(1), 2007.

[22] M. Störzer and C. Koppen. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, September 2004.

[23] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE*, 2001.

[24] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *ACM TOSEM*, 20(2), 2009.

[25] T. Young. Using AspectJ to build a software product line for mobile devices. Master's thesis, UBC, 2005.