Aspect-Oriented Reengineering of an Object-oriented Library in a Short Iteration Agile Process

Adrian O'Riordan
Computer Science Department, University College Cork, Cork, Ireland
E-mail: a.oriordan@cs.ucc.ie

Keywords: aspect-oriented software development, reengineering, agile development, software metrics, refactoring

Received: June 24, 2011

Aspect-oriented reengineering aims to modularize crosscutting concerns in an existing system using a new abstraction called an aspect. Code concerns may be tangled and scattered throughout an existing code base thus hampering maintenance. This paper describes the reengineering of an object-oriented software library called GEF using aspect-oriented techniques as an integral activity in an agile process. Graph Editing Framework (GEF) is a medium-sized open source Java library for the construction of graph editing applications. We evaluated both the original and reengineered code by applying a set of appropriate software metrics to measure to what extent aspect-oriented refactoring affected modularity attributes such as coupling, cohesion and complexity. To mirror a real world setting, analysis, re-design, and semi-automated refactoring was performed in three-week iterations typical of agile development using tools freely available on the Eclipse platform. We found that only marginal improvements in modularity were possible in that timeframe and argue that fully-automated aspect mining and refactoring tools are needed to bolster aspect-oriented reengineering.

Povzetek: Članek opisuje predelavo knjižnica z agilnim aspektno usmerjenim programiranjem.

1 Introduction

Aspect-oriented software development (AOSD) promises to improve the modularity of software by the separation of concerns into aspects during system development. This paper presents a study of aspect-oriented reengineering involving the analysis, re-design and refactoring of an existing medium-sized object-oriented library. This was done in a way that was authentic or faithful to industry practice where refactoring is an integral part of agile methods [1]; analysis, re-design, and refactoring are performed in short iterative cycles using tools widely available. Recently developed research tools in automated code transformation and aspect mining that are as yet not common in industry were thus not employed. The time spend in the refactoring phases of development was not changed from that commonly spend in conventional object-oriented refactoring despite the introduction of aspect technology.

We carried out the aspect-oriented refactoring or *aspectization* in a semi-automated manner as part of an agile development process. Agile methods have become popular and already incorporate refactoring in their development process and hence are a suitable approach for introducing aspect-oriented refactoring into a reengineering process. We employed the AspectJ language and associated development tools for refactoring. The two developers were experienced in Java development but only recently familiar with AOSD and AspectJ. We applied a metric suite to both the original and reengineered library, comparing the two sets of results in order to establish any improvements in the areas of reduced complexity, reusability, and maintainability. Conclusions are drawn on the efficacy of this approach.

1.1 Aspect-oriented software development

A reality of modern software is the requirement for continuous change. This change can be instigated externally by the discovery of bugs or changing customer needs or internally to an organization for technological or institutional reasons. Software evolution and maintenance is hampered by the types of decomposition used in coding and design: separation of concerns is a long standing challenge in software engineering [2]. A key problem in software evolution is that software designs tend to have a dominant kind of modularization. This could be feature-based (e.g. transactional) or paradigmatic (object-oriented). But changes that affect a particular feature or concern (such as security for example) may favour an alternate decomposition [3]. In particular, the limitations of object orientation are now becoming more apparent – such as in feature segregating or in applying domain-specific knowledge [4]. AOSD is a technology that addresses the separation of concerns in software at the code level.

The concept of an *aspect* originated at Xerox PARC in the form of aspect-oriented (AO) programming [5], and has gone on to receive significant attention in the software engineering research community [6]. AOSD developed out of work in object-oriented (OO) programming, reflection, and the meta-object protocol [5]. The aim of AOSD is to modularize crosscutting concerns in a system to manage the structural

relationship between representations of a concern. These code concerns or areas of interest can be scattered and tangled (intermixed) throughout the design and implementation; common examples include error handling, logging, and security. Concerns can relate to functional or non-functional requirements. Crosscutting concerns are claimed to make systems difficult to maintain, increase the complexity of the system and reduce the reusability of the code [7]. By applying AO techniques, these concerns can be put into separate modules called aspects, untangling them from each other. Though the AO approach was developed as a programming method, it has been extended to encompass more stages of the software development lifecycle [8].

AOSD tackles areas not addressed in a purely object-oriented OO approach to software development. For existing software to benefit it will be necessary to support the migration of legacy systems to AO solutions. Just as the adoption of OO software development lead to the need to reengineer legacy systems, as for example in [9]; the wider adoption of AOSD will require a similar effort. Laddad advocates a safe adaptation path for AOSD where AO refactoring is applied before AOSD is exploited from a project's inception [10]. There is less experience of applying AOSD in industry and few experience reports published as yet, see Section 5.

1.2 Overview of paper

The paper is structured as follows. Section 2 introduces background material on AO programming, the GEF library and the metric suite. Section 3 presents the reengineering implementation. Section 4 contains the evaluation. The paper finishes with a summary and conclusions.

2 Background

2.1 Aspect-oriented programming in AspectJ

AO programming introduces a number of unfamiliar concepts to programmers. These concepts offer additional functionality to assist with the modularization of crosscutting expressions by encapsulating a concern in one place that would otherwise cross existing units of modularity such as class, subprogram and package. We follow the formulation and terminology of AspectJ throughout this paper.

AspectJ is described as a seamless extension to the Java programming language [14]. AspectJ is free open source software available under an EPL (Eclipse Public License). The major Java extension called an aspect has a Java class style syntax. All legal Java programs are upwardly compatible with AspectJ and all AspectJ programs run on any Java Virtual Machine. The process of linking classes and aspects together is called weaving. In the case of AspectJ, this produces executable bytecode. The bytecode produced by the AspectJ compiler should be comparable to the bytecode produced by a Java compiler used on an equivalent (scattered and tangled) Java implementation [11]. The AspectJ Development Tools (AJDT) [1] provide Eclipse platform based tools for editing, building and debugging AspectJ programs. Whereas Eclipse has good support for AOSD, other IDEs have lagged behind. Alternatives to AspectJ include Hyper/J but AspectJ is by far the most widely deployed example of aspect technology at present.

Here is a brief summary of the operation of AspectJ; see [10] or [12] for a more detailed description. An aspect is a new unit of modularity providing encapsulation and abstraction and allowing tangled or scattered code to be removed from classes while still maintaining overall functionality. *Join points* are events that occur during the runtime execution of a program, for example each time a method or a constructor is called or a variable created. Each such run-time event is a separate join point visible to aspects during program execution.

A *pointcut* is used to identify, by matching, join points of interest. Examples of pointcut designators are *call*, *execution*, *target*, *this*, *get*, *set*, and *args*. There are both named pointcuts and property-based pointcuts that can have wildcard expressions. Pointcut expressions can be created with the &&, || and ! Java logical operators. Pointcuts can also expose contextual information at the join points that they match. Once a pointcut has matched a join point, advice specifies what is to occur.

Here we briefly explain the function of the designators that are used in the example code in Section 3.The *execution* designator picks out each method execution join point and *target* picks out each join point where the target object is of a specified type. The *within* designator limits the lexical scope of the join point and the *this* designator checks runtime type. A *cflow* picks out a join point within the dynamic context of another.

Advice is unnamed as it is implicitly invoked. There are three main types of advice. *Before advice* is advice that executes before a join point whereas *after advice* executes immediately after a join point. *Around advice* runs in place of the join point and is the most flexible type of advice since it can change contextual information. In general terms, an AO programming implementation is characterized by its join point model which dictates the location of joint points (where advice can run), quantifies joint points (how they are matched) and specifies what to do (for example run advice).

AspectJ also has *inter-type declarations (ITDs)*, formerly introductions. ITDs are declarations that affect a program's static structure. They are mainly used to provide definitions of fields and methods within an aspect on behalf of other classes. ITDs can be viewed as enabling open classes allowing structural additions. Note that aspects intercept base code without needing to modify it. This thus makes AO refactoring possible even when the base code cannot be changed.

2.2 Reengineering

Reengineering aims to restructure legacy software. Without comprehensive design specifications maintaining legacy code can be a major burden. Even where extensive documentation exists, reengineering and

software evolution can entail making changes throughout a software system, and has been found to be both difficult and tedious [13]. Reengineering is the examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of this new form [14]. Reengineering generally consists of some reverse engineering or design discovery (often to achieve a more abstract representation) followed by restructuring. Existing OO reengineering does provide some techniques for dealing with tangled code. Refactoring [15] enables OO code restructuring and is an integral part of agile software development methods [16]. Agile methods such as Extreme Programming advocate a culture of continuous reengineering [17].

Many IDEs, such as Eclipse, now have support for a semi-automated refactoring process. Code refactoring includes techniques for renaming, decomposing, composing, relocating, and abstracting program code elements such as identifiers, methods, and classes. Two examples of code refactoring include extracting a method, and converting conditional code into polymorphic code. The aim is to improve quality measures such as "understandability", reusability, and maintainability; not to fix bugs or introduce new features.

But there are limits to the application of OO refactoring and the extent to which conventional refactoring can disentangle code [18]. To give just one indicative example, behaviour can be delegated to a separate class, but new problems can consequently be created because delegation decreases cohesion and adds additional components [19]. In addition, there are scenarios where it is very difficult to separate out a concern using conventional OO techniques, thus impacting ease of maintenance. This may lead to updates being required for unrelated modules for a minor change.

2.3 GEF library overview

The object-oriented software library that was reworked is GEF (Graph Editing Framework), a medium-sized free open source Java library for the construction of graph editing applications [2]. GEF is not a complete drawing program but it supports the construction of custom drawing programs. ArgoUML[3] is a popular open source UML modelling tool built using GEF. GEF (Version 0.12.3) was chosen for the reengineering project for two main reasons: (i) as a medium-sized application it is nontrivial but manageable: and (ii) because it is already well-designed using conventional OO design, any reengineering can focus on the benefits of AO restructuring.

Figure 1 shows screen captures of a simple demo application that uses GEF. GEF is designed using the Model-View-Controller architecture separating the graph models from the display information in Java SWING. GEF was developed to be easy to use and extend without modifying the underlying framework. A flexible Node-Port-Edge graph model is employed for drawing objects. Briefly stated GEF supports selection, grouping, layering and views but not zooming and undo. GEF specifies data as generic properties using JavaBeans. XML-based file formats are employed based on the PGML standard. GEF is a Java counterpart to graph editing libraries such as Unidraw (C++) and HotDraw (Smalltalk).

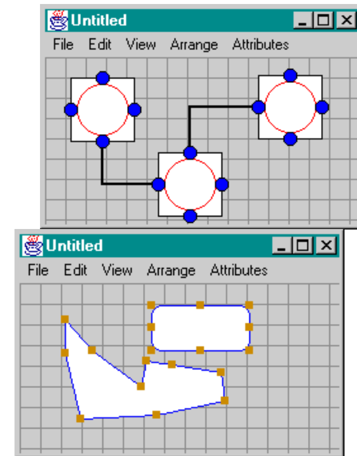

Figure 1: Screenshots of GEF demo application.

The most important classes are now briefly introduced; many of these are referred to in the refactoring in Section 3. Editor is the central class of the Graph Editing Framework. There is one instance of Editor for every diagram that is displayed on the screen. Editor does not handle input events, or modify a diagram; instead it passes events and messages to supporting objects. An Editor has a LayerManager which manages a stack of Layers. Layers contain the objects to be drawn, which are called Figs. Layers group Figs into transparent overlays. Figs are drawable objects that can be shown and manipulated in the Editor such as rectangles, lines, circles, and text. FigGroup is the class for groups of Figs to be treated as single items. When a Fig is selected the SelectionManager holds a selection object. Selections are objects used by the Editor when the user selects one or more Figs. Selections indicate the target of the next command. The behaviour of the Editor is determined by its current Mode. The Editors ModeManager keeps track of all the active Modes. Modes interpret user input events and decide how to change the state of the diagram. Examples of Modes are ModePopup which deals with right mouse button events and shows a popup menu and ModeSelect which allows one to select one or more figs. Cmd is an abstract class for all editor commands. Classes starting with Cmd (CmdSelectAll, CmdCopy, etc.) are classes that define a doIt() method that performs some action in the Editor. In total GEF consists of 302 classes and 30835 lines of code, broken up into 14 different packages. There is little documentation apart from the Javadoc API. Figure 2 shows the major classes of GEF in a reverse engineered MVC architectural design view that serves as the starting point for the re-design.

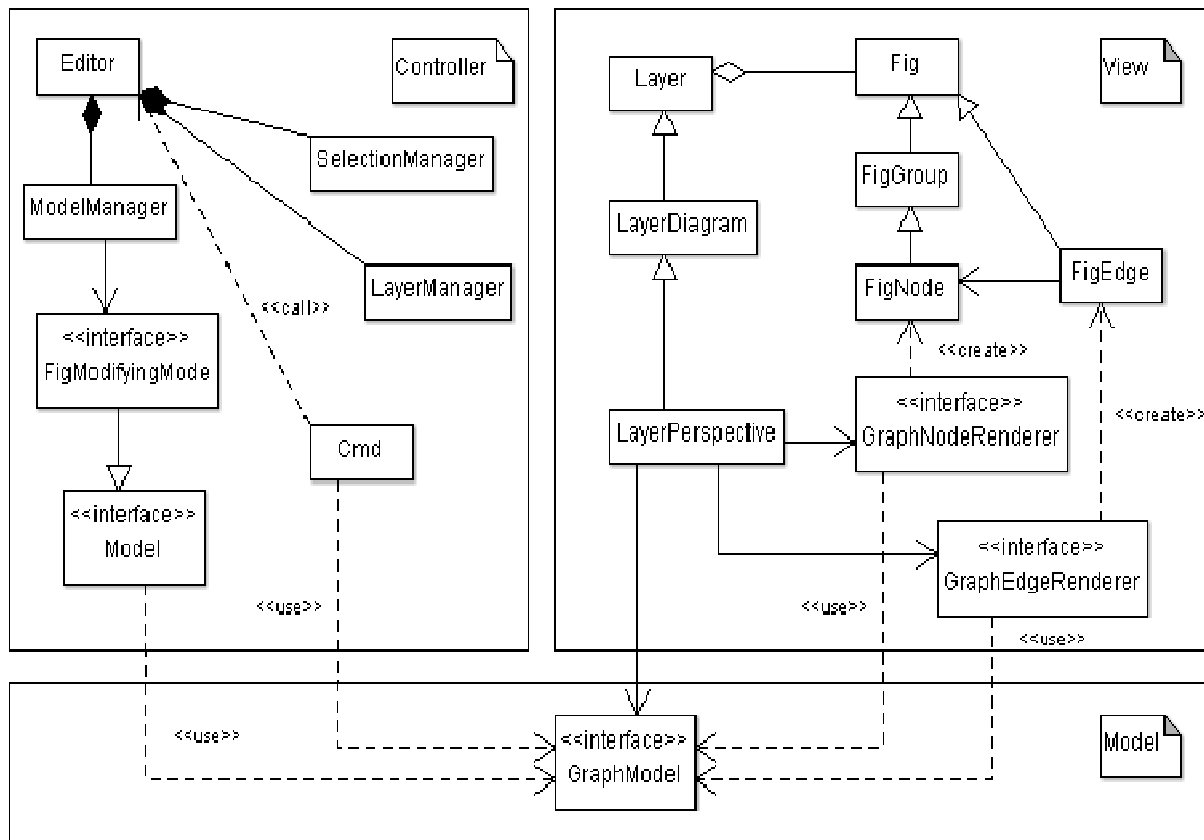

Figure 2. Reverse Engineered MVC Design of GEF.

2.4 Software metrics employed

Metrics for assessing modularity cannot be analyzed independently of other metrics of program quality. For example, a software system implemented as a single module has no inter-module communication but may be deficient in many other regards. Many software metrics have been devised based around the concepts of coupling, cohesion and complexity. In the broadest sense modularity relates to API compatibility, testability, maintainability and extensibility. A summary of the metrics employed in this study is given below. We employed Aopmetrics[4], an open source metrics tool for OO and AO programming. It provides AO extensions to many common OO metrics which can be used to measure the code base and make predictions on reuse and maintenance. Most of the metrics fit into the categories of size metrics, coupling metrics, cohesion metrics and complexity metrics, comparable to the Chidamber and Kemerer (C&K) OO metrics [20]. Additional package dependency and aspect-specific metrics are also present. Note that we use the Java terms *class* and *method* in the following summary descriptions where the Aopmetrics documentation has the terms *module* and *operation*.

Size metrics

Lines of Class Code (LOCC): LOCC gives the total non-blank and non-commented lines of class code.

Complexity metrics

Weighted Operations per Module (WOM): WOM counts the number of methods in a given class, capturing the internal complexity of a class which is an indicator of how much time and effort is required to maintain the class. Classes with a large number of methods may be too complicated or very application specific thus limiting reuse. Response for a Module (RFM): RFM of a class is the number of methods and advices that potentially can be executed in response to a message received by the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated.

Coupling metrics

Coupling on Method Call (CMC): CMC is the number of classes or interfaces declaring methods that are possibly called by a given class. Usage of a high number of methods from many different classes indicates that the function of the given class cannot be easily isolated from the others. Coupling between Modules (CBM): CBM is the number of classes/aspect or interfaces declaring methods or fields that are possibly called or accessed by a given class. Excessive coupling between classes is detrimental to modular design and prevents reuse. Depth of Inheritance Tree (DIT): DIT is the length of the longest path from a given class/aspect to the class/aspect hierarchy root. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour.

Afferent Coupling (Ca): Ca measures the number of classes outside a package that depend on classes inside the package [21]. Efferent Coupling (Ce): Ce measures

the number of classes inside a package that depend on classes outside the package.

Cohesion metrics
Lack of Cohesion in Operations (LCO): LCO measures the number of methods within a class that access one or more of the same attributes. Low LCO is desirable.

Package dependency metrics
Normalized Distance from Main Sequence (D): D is the distance of a package from the idealized line *Abstractness + Instability = 1* where *Abstractness* is defined as the ratio of the number of abstract classes to the total number of classes in the package and *Instability* is the ratio of efferent coupling (coupling outside package) to the total coupling. D is an indicator of a package's balance between abstractness and stability. This metric has a range $0 < D < 1$, where a 0 indicates ideal package design.

Aspect-oriented metrics
Crosscutting Degree of an Aspect (CDA) CDA is the number of classes affected by the pointcuts and by the inter-type declarations in a given aspect. CDA measures all classes possibly affected by an aspect. High values of CDA are usually desirable.

3 Reengineering the GEF Library

3.1 Adoption risks and process overview

There is significant adoption risk associated with AO technology: (i) lack of tool support; (ii) lack of education; (iii) implementation issues; (iv) unpredictable behaviour due to code injection; and (v) security issues [14]. Many of these issues will dissipate as tools and methods mature and gain wider acceptance. Issue (iv) is of particular relevance to AO reengineering as existing code bases with agreed upon contracts can be altered. Supporting processes and techniques, such as embodied in test-driven development help ensure unanticipated behaviour is not introduced. Introducing any major new technology has been found to cause an initial decrease in programmer productivity [22]. Laddad in [10] recommends a cautionary approach for AOSD adoption first employing simple AO techniques for common concerns such as logging and exception handling, to be followed by the more complex techniques for trickier concerns. Applying AO techniques to legacy systems can pose difficulties for various reasons: large code size, lack of documentation, complexity and inconsistencies of implementation and the need to preserve behaviour. A recent review endorses an incremental adoption path [23].

As yet there is no established process for software reengineering. Organisations have typically adapted their standard development process; for example, NASA [24]. We employed an agile process of short development iterations where refactoring is a major component. Adopting an agile approach, two engineers worked in four approximately three-week, development iterations consisting of analysis, design discovery and two iterations of AO code refactoring. Agile methods, such as Scrum and Extreme Programming, use refactoring to improve software quality and enhance project agility. The typical approach in agile development is to first write tests, by means of automated unit testing, which are subsequently used to verify that code transformations are behaviour preserving.

A refactoring process suitable for reengineering is described by Kataoka et al., where refactoring is done in iterations or "clumps" [25]. Pizka took a similar approach of short iterations of discovery, application and test [26]. In particular, similar approach we followed was: (i) Identify code to be refactored; (ii) Determine which changes to apply and to where; (iii) Write tests; (iv) Apply refactoring; (v) Assess effects and check that change is behaviour preserving. This mirrors the typical process required to manage incremental change in OO refactoring: determine change, locate relevant code and determine the change's extent, and carry out impact analysis.

For each new aspect we introduced we examined relevant source code call method calls, constructors, and blocks of code. Once an aspect was introduced, its functionality and purpose were reviewed and adjustments made to further refine how it interacted with and contributed to the existing classes, as well as adjusting the classes in the library that the newly created aspect was now advising. An aspect that starts off interposing a single class can be used to interpose multiple classes. This process was repeated for each crosscutting concern that was identified. Static and dynamic tests were performed to guarantee that software behaviour was preserved, that is to ensure that for the same set of input values, the resulting set of outputs were the same before and after refactoring. Stronger notions of behaviour preservation are needed for domains such as real-time and embedded systems where performance and other properties such as safety could be affected. This was not an issue with GEF.

3.2 Refactoring GEF

We carried out two iterations of AO refactoring using the semi-automated techniques. In the first refactoring iteration we concentrated on basic concerns and solutions within our limited time window. Common AO refactorings that we used included *Extract Feature into Aspect* and *Extract Fragment into Advice* [27]. We measured the restructured software after this phase. In the second iteration we did further refactoring primarily based on the AO implementation of established design patterns and re-ran the metric suite so the modularity of the software was again measured at this final stage. Design patterns are an attractive proposition for software design but while the benefits to the design are well documented [28], their implementations "tend to vanish in the code" [29], failing to "capture the concern explicitly" in the code [30]. AO implementations of design pattern retain explicitness while offering the desired benefit.

AO code refactoring differs from and is more pervasive than he conventional OO refactoring techniques mentioned in the introduction [31]; for example, while extract method simply moves code into a new method replacing it with a method invocation, using aspects you can take an additional step and take out those invocations in the source code altogether. You can make changes not possible with just a Java compiler such as moving out try/catch blocks into a separate aspect. Many of the guidelines and practices in OO refactoring, carry forward to AO refactoring. As stated in [32], AO refactoring "augments and not replaces conventional refactoring." AO refactoring techniques have been developed to modularize exception handling, concurrency, lazy initialization, contract enforcement, and a number of other design constraints. Catalogues of AO *refactorings* have been developed [27, 31, 33, 34]. These are often described in the template format popularized by the design pattern community.

We used the aforementioned Eclipse tools, AJDT and JUnit for writing unit tests. The AspectJ graphical structure browser and the Visualizer allowed us to identify concerns without the use of a dedicated aspect mining tool. Tool-supported refactoring can greatly reduces the effort of manually scanning and changing code. We return to the issue of automated aspect mining and automated refactoring in Section 5. Note that we used only a subset of the AspectJ language features. Indeed, as of 2010, most industrial applications of AOSD have used only basic features [23, 35]. We followed the guidelines given by Colyer [16] where pointcuts are named and individual pointcut definitions are kept simple. Named pointcuts can thus be reused. We placed all pointcuts in an aspect next to the associated advice. In AJDT you can handily associate run-time tests with each item of advice.

In particular the following separate of concerns (SoCs) were addressed wherein one or more aspects were introduced to deal with each.

- SoC1: Exception Handling
- SoC2: Logging
- SoC3: Notification Services
- SoC4: Event Handling
- SoC5: Design Pattern Concerns
    - Composite
    - Strategy
    - State

3.2.1 AO refactoring iteration 1

A summary of the five aspects introduced in the first iteration of refactoring are given next. *ThrowableException* is an aspect introduced to deal with SoC1. Following are additional aspects dealing with SoC2, SoC3 and SoC4 in turn.

ThrowableException aspect (SoC 1)
This is the simple aspect. Calls to printStackTrace() are made by some catch clauses in the original code. Such code snippets occur in multiple packages. We created a new package called exception, modularizing the crosscutting code into an aspect called ThrowableException. This will be single aspect instance – by default all aspects are singletons. Pointcut expressions are created matching join points that can occur in the Java source code. Around advice executes at the matched join points. As the program executes, the pointcuts match events in the runtime of the application triggering a stack trace method to execute. This allows duplicated source code to be removed, providing benefits such as improving the readability of the base code, having the exception throwing all in one place, and supporting future additions which may need to implement calls to printStackTrace(). This example uses the *execution*, *target* and *args* pointcut designators. The *args* designator used here captures contextual information, in this case the arguments passed to methods at an execution joinpoint.

```
package exception;

public aspect ThrowableException{

pointcut printingStackTrace(Throwable aCause):
    execution (* printStackTrace()) &&
target(aCause);

// other pointcuts elided

void around(Throwable cause)
:printingStackTrace(cause){
    proceed(cause);
    if (cause != null){
       System.out.println("Caused by:");
       cause.printStackTrace();
    }
}
//other advice elided
}
```

ExceptionHandler aspect (SoC1)
Exception handling occurs throughout a number of classes in the util package of the GEF library. Within this package we have modularized all try-catch clauses into a second aspect called ExceptionHandler. Exception handling also occurs in classes in other packages of the library but its use is applied in an inconsistent manner and so it was not possible to modularize into this aspect. Within the util package there existed a number of try-catch clauses in classes tangled with other logic in the class. We moved all this exception handling code into the aspect.

LoggingCalls aspect (SoC2)
Logging is used by a number of classes in the GEF library for debugging purposes. Logging is not applied uniformly throughout the library but instead is used on an ad hoc basis in a number of different classes.

It was possible to modularize checks that were done before a message was logged. Before a message is logged with debug priority (Log.debug("message")), a check is made to ensure that debug logging is enabled (Log.isDebugEnabled()). If the result of this check is true then the message is logged, if the result is false then logging is ignored. This check occurs in 70 different locations throughout the library, in a different classes and packages. Since this check is not a primary concern of

the classes in which it occurs, it was moved to an aspect. The aspect contains a pointcut that matches any join points that occur when a call is made to Log.debug() in GEF. When a match is made, contextual information is extracted from the join point; the Log object is extracted and made available to the aspect. A check is then made using around advice, which results in control returning to the join point if debug logging is enabled. If debug logging is not enabled, messages are not logged and control returns to the code after the join point.

PropertyChangeHandler aspect (SoC3)

In GEF the Globals class stores global information that is needed by all Editors. Within the Globals.java class, listener notification is implemented. A hashtable is created which keeps track of a number of PropertyChangeListeners for Figs. It allows for PropertyChangeListeners to be added to Figs. Any changes to the properties of a Fig will result in a notification being sent to its listeners. A Fig can have up to four listeners. There are five methods implemented in the Globals class that manage these propertyChangeListeners.

The methods for managing the properties are not scattered across the library but we decided to modularize these methods since they are specific to Figs and are not the primary concern of the Globals class. By modularizing them into an aspect, PropertyChangeHandler, the code in the Globals class becomes less complex and more robust if changes need to be made to the way listeners are handled.

```
public aspect PropertyChangeHandler{

private static Log Globals.LOG =
    LogFactory.getLog(Globals.class);
private static Hashtable Globals._pcListeners =
    new Hashtable();
private static PropertyChangeListener
    Globals.universalListener = null;
public static int Globals.MAX_LISTENERS = 4;

    public static void
Globals.addPropertyChangeListener (Object
    src, PropertyChangeListener l){
    PropertyChangeListener listeners[]=

(PropertyChangeListener[])_pcListeners.get(src);
if (listeners == null){
    listeners = new
    PropertyChangeListener[MAX_LISTENERS];
        _pcListeners.put(src, listeners);
}
for (int i = 0; i < MAX_LISTENERS; ++i)
    if(listeners[i] == null) {
        listeners[i] = l;
        return;
    }
}

public static void
Globals.addUniversalPropertyChangeListener
    (PropertyChangeListener pcl) {
    universalListener = pcl;
}

public static void
Globals.removeUniversalPropertyChangeListener(){
          // code cut for brevity
}

public static void
Globals.firePropChange(Object src, String
propName, boolean oldV, boolean newV){
    firePropChange(src, propName, new
        Boolean(oldV), new Boolean(newV));
}
// overloaded methods cut for brevity
}
```

This example also shows how the observer pattern is quite naturally implemented in AO programming. Also an alternative option, OO refactoring involving delegation, is not an attractive option here because of the added level of indirection and complexity. The trade-off between using and not using inheritance or delegation is an on-going area of debate. Empirically measuring generalization costs against reuse savings has proved difficult. An interesting proposed solution involves a cost-benefit approach to develop a suitable metric [36].

UseActionEvents aspect (SoC4)

There are classes in the GEF library, UseReshapeAction, UseResizeAction and UseRotateAction, which implement almost identical event listening methods. These classes deal with allowing an Editor to perform certain actions on groups of objects that are currently selected. These actions are a resize action, rotate action and reshape action. Since this is an area of the library where there may possibly be future additions of new classes that provide additional actions, we modularized this duplicated code which is crosscut among these classes into an aspect.

A new aspect called UseActionEvents was created which matches the execution of any actionPerformed() methods in the above classes. When the pointcut defined in the aspect matches a call to this method during the runtime execution of the class, control is passed to the aspect, which then executes some event listening logic depending on where the call originated from. Once the aspect is finished executing, control is passed back to the class.

```
public aspect UseActionEvents{

pointcut
handlingActionEvents(UseReshapeAction aReshape):
    execution (public void actionPerformed(..))
&& target(aReshape);

// other pointcuts elided

void around (UseReshapeAction reshape):
    handlingActionEvents1(reshape){
    Editor ce = Globals.curEditor();
    SelectionManager sm =
        ce.getSelectionManager();
    Enumeration sels = ((Vector)
        sm.selections().clone()).elements();

    while (sels.hasMoreElements()) {
        Selection s = (Selection)
            sels.nextElement();
        if (s instanceof Selection &&
        !(s instanceof SelectionReshape)){
            Fig f = s.getContent();
            if (f.isReshapable()){
                ce.damaged(s);
                sm.removeSelection(s);
                    SelectionReshape sr = new
```

```
                SelectionReshape(f);
            sm.addSelection(sr);
            ce.damaged(sr);
        }
      }
   }
}
//other advice elided
}
```

3.2.2 AO refactoring iteration 2

A summary of the four additional aspects introduced in the second iteration of refactoring are given in the following subsections. The first of these related to the repaint method and addresses SoC3. The remaining three new aspects address SoC5. Code samples of the reengineered library are included for some of these.

Repainting (SoC3)
First we give a brief description of how the repainting of graphical objects takes place in GEF. Mouse events, screen damage, or changes to a figure's boundary necessitate repainting the screen. Damage is stored as a list of rectangles. This is part of the RedrawManager class's responsibility as well as determining the object under a given mouse point. In GEF, a Layer class can dictate the redraw order of a group of Figs. A Layer is responsible for notifying all dependent layers of changes. Different layers can be hidden, locked, or grayed out independently. A complex notification service maintains state. We introduced a new Repaint aspect that provides an aspect-based implementation of this notification mechanism. This is again based on the Observer pattern and operates similar to the PropertyChangeHandler described in Section 3.1. This necessitated moving and reworking code in RedrawManager.

Composite pattern for handling FigGroups (SoC5)
FigGroup has methods that perform various actions, such as setting and removing properties on all of the Figs in a FigGroup. In the original library different iterators process the list of Figs for each of these. We introduce an aspect to perform these updates. Note that the update operation requires contextual information in the form of the particular type of update operation.

```
static aspect UpdateAllFigs{
   pointcut updateOp (FigGroup fg):
   execution(   *   FigGroup.*(..))   &&   this
(FigGroup) && within (FigGroup);

pointcut FigGroupOperation(FigGroup fg):
   cflow (updateOp);

   // advice elided
   }
```

This example uses the *this*, *within* and *cfow* pointcut designators. The *cflow* designator specifies that the pointcut is in the control flow of each join point picked out by the updateOp pointcut. The pointcut expression with the execution designator matches all executions of any FigGroup method.

Strategies for different commands and state pattern for changing behaviour of Editor depending on FigModifyingMode (SoC5)
Depending on the context the various subclasses of Cmd can be used to perform a suitable action. This part of GEF isn't fully developed as operations such as Undo are not supported. During refactoring the various subclasses of Cmd were removed from the code simplifying the source class design by means of an aspect-oriented implementation of the strategy design pattern [33][37]. We attach advice corresponding to each command type as described in [33]. After advice is used to modularize the various states of FigModifyingMode. This has the advantage of localising future changes since this is extensively used.

4 Evaluation

The following sections presents the metrics after reengineering of the library was completed. Due to the large number of classes involved and the scattered nature of some concerns, for each metric we took average values for the entire library, to give an indication of what effect reengineering had on the library as a whole, with the exception of the *Lines of Class Code (LCC)* metric and the *Weighted Methods per Class (WMC)* metric.

4.1 Evaluation results

Table 1 gives the coupling and cohesion results.

Metrics	Original	Re. Iter 1	Re. Iter 2
CMC	3.205	3.140	3.013
CBM	3.246	3.181	3.126
DIT	1.383	1.383	1.383
Ca	14.81	14.67	14.67
Ce	10.90	10.90	10.90
LCO	117.6	117.6	117.9

Table 1: Coupling and cohesion results.

The coupling and cohesion results did not show dramatic changes between the original and the reengineered code, but the changes do give indications of the effect that the introduction of aspects had. Overall, there is a small reduction in coupling. The *Coupling on Method Call (CMC)* metric showed approximately a two and six percent average decrease in coupling for refactoring iteration 1 and 2 respectively. (Aopmetrics gives results to seven digits of precision but in all the tables here these are rounded down to four. The percentage increases and decreases are rounded to the nearest percentage.) The *Coupling between Modules (CBM)* metric showed a small average reduction of two and four percent between the original and reengineered library. *The Depth of Inheritance Tree (DIT)* metric remained the same for both versions of the library due to the fact that the introduction of aspects did not affect the class hierarchy in the way that subclassing would through OO refactoring. This observation has been previously published [38]. There were small reductions in *Afferent Coupling (Ca)* whereas *Efferent Coupling (Ce)* remained the same. There was a slight increase in the *Lack of Cohesion in Operations (LCO)* metric between the original and reengineered library. Generally high cohesion is a desirable property and so a reduction in lack of cohesion would have been the preferred result.

However, the increase is relatively minimal, and since *LCO* is a measure of the number of methods within a class that access one or more of the same attributes, the use of some inter-type declarations in aspects may have contributed to the increase.

Metric	Orig.	Re. Iter 1	Re. Iter 2
LCC	30835	30422	30355
RFM	3.246	3.181	2.952
WOM	7158	7023	7010

Table 2**:** Size and complexity results.

Table 2 has results related to size and complexity. The metrics Weighted Methods per Module (WOM) and Response for a Module (RFM) are good indications of both the internal complexity and overall complexity of classes. The RFM decreased for the reengineered version by approximately four and ten percent which indicates a small reduction in complexity. The LCC metric indicated a small reduction in code size. This small reduction is due to the removal of replicated code into aspects as well as the movement of some methods and fields.

The very slight increase in the LCO metric is not significant because the overall change in this metric was relatively small. Also there are uncertainties with respect to the level of confidence that can be put in this metric due to the varied results it has displayed in other studies; see Section 6. It is best to consider the results of a set of metrics rather than just one metric in isolation. The results obtained for RFM and WOM support claims of a reduction in complexity, which may have a knock on effect for encouraging reuse and simplifying maintenance. The D metric also provides reassurance that the reengineering has not caused any major stability issues in the library.

Table 3 below shows results for package stability and dependency where there was no significant movement.

Metrics	Orig.	Re. Iter 1	Re. Iter 2
D	0.426	0.427	0.427

Table 3: Package dependency results.

The crosscutting degree metric (CDA) displayed in Table 4, is not applicable for purely OO systems but comes into play when aspects have been used.

Metrics	Orig.	Re Iter 2	Re. Iter 2
CDA	0	44	89

Table 4: Aspect-oriented results.

5 Discussion

The use of AO techniques to reengineer the GEF library using semi-automated techniques in a tight timeframe proved only marginally beneficial to the overall design quality of the library in most areas. The results after applying the metrics support AO programming claims of reducing complexity and coupling but only to a small degree. We believe this was due to the fact that only a limited number of refactoring can be achieved in six weeks.

Similar negative results have been obtained from experiments on conventional refactoring; see for example [26], which used a medium sized Java code base and also a tight developer timeframe. Wilkin et al. also report disappointing results [39]. In another refactoring experiment, Bourquin and Kellen [40] note that code size reduced by ten percent but only after seven months of refactoring, though this involved a much larger code base (140 KLOC of Java) but the team size is not specified. Previous experience of more extensive reengineering, where a software system is modified by above 20 to 25 percent, has been found to be counterproductive [41]. Chen et al. has data on the human effort of OO refactoring, although this was restricted to exception handling [42]. 41 man-hours were spent refactoring 14 KLOC of Java with 371 LOC being modified. They deem the effort to be worthwhile based on a cost-benefit analysis calculated as the estimated savings in maintenance cost minus development costs (man-hours by engineer's pay per hour).

Though there are a number of case studies on aspect-oriented refactoring, see Section 6, unfortunately there is little concrete information provided in how many man-hours were involved in the various tasks. This early-stage work has so far, understandable, concentrated on methods and tools.

Some difficulties we encountered while reengineering are worth mentioning. A lot of time was spent analysing and re-designing GEF, for example identifying sites where an AO approach could be taken. Possibly because the system is a library as opposed to an actual application, a lot of classes were already relatively independent and modularized, limiting where aspects could be used. In many applications there are stand-out crosscutting concerns such as database access and security/authentication that are good candidates for AO refactoring. Persistence is another common concern that is amenable to an AO solution [30] that did not feature in the GEF library. In parts of the library it was difficult to cleanly remove all the code associated with some concerns such as logging. During the modularization of exception handling in the util package, additional lines of code and contextual data had to be extracted from the join point into the aspect, which was not ideal.

Here we briefly discuss two limitations of our methodological approach. While we did some we did not do widespread OO refactoring prior to the AO refactoring. It has been stated that initial code restructuring such as via OO refactoring can aid subsequent AO refactoring [43]. Capturing some concerns as aspects may necessitate restructuring of the base code to expose suitable join points. Second, we did not measure stability in the face of actual changes. Greenwood et al. performed an extensive empirical study of design stability in the face of system changes that are typically performed during software maintenance tasks finding that AO implementations tend to have a more stable design than purely OO implementations [44].

Tools to automate AO reengineering have begun to appear but are still at the research stage of development. Aspect mining techniques are vital to automate the aspect discovery phase. Kellens et al. provide a comprehensive survey of emerging aspect mining techniques [45]. Different approaches are being tried to help identify aspect candidates such as text analysis, dynamic program analysis, code slicing and natural language techniques. Research tools such as DynaAMiT, DelfSTof, Dynamo, and AOPMigrator have recently been developed [45][46]. Work is needed to make these more scalable, more usable and more widely known so as to transfer the technology to industry.

Fully-automated refactoring is the second major component needed to enable full automation. In automated refactoring, refactoring consists of program transformations that satisfied specified preconditions. At present AO refactoring is mostly done by hand or in the semi-automated way because of the immaturity of automated AO refactoring support tools and the fact that those that do exist cannot guarantee they are behaviour preserving [45]. IDEs such as Eclipse currently support a user-guided (or semi-automated) approach but a lot of human effort and expertise is still required. Research in fully automating OO refactoring is actively ongoing.

A property of software that can be affected by any type of refactoring is performance. Generally AO programming has been found to have a negligible effect on performance [10]. Some research has even shown unanticipated performance improvements after OO refactoring [47]. We ran the original and refactoring GEF Demo application are there was no noticeable performance differences.

5.1 Related studies of aspect-oriented reengineering

The majority of empirical studies have shown that applying AO concepts to applications can improve modularity and provide benefits in the areas of reduced complexity, maintainability and reusability but most of these studies don't explicitly state how much effort went into the reengineering.

The very small reduction in lines of code we observed is in line with similar studies [48, 49, 50]. Studies of the reengineering of AO software systems, such as those by Kendall [19], have shown improvements in modularization. This study entailed role modelling of intelligent agent protocols and concentrated on refactoring inter-agent communication and agent conversation/negotiation. Note that Kendall's reengineering used both traditional OO refactoring as well as AO refactoring. Work in the areas of exception handling [48, 49] have shown that the use of aspects helped reduce code tangling and loosen class coupling. Unlike our work, these two studies were restricted to one functional area, exception handling. Evaluations of AOP programming for real-time systems [50] also showed improved modularity for crosscutting concerns. Mixed results were obtained in a project reengineering the Hypercast system for multicast overlay networks [51]. The original Java implementation had 300 classes and was redesigned first using common AO programming methods, pointcut descriptions and advice. They found this approach led to programs that were "unnecessarily hard to develop, understand and change." They repeated the experiment with abstract interfaces that expose pointcut descriptors and impose contracts and found this easier and led to a clearer design. Zhang and Jacobson found a 22 percent decrease in coupling in reengineered middleware [52]. A study by Madeyski and Szala was inconclusive [53]. While other studies show a desirable change for the LCO metric [48], there are also studies where lack of cohesion increased [49]. This may indicate limitations of usefulness of this metric in AO systems or possibly calls for modifications on how the metric is calculated.

Using software metrics to mine aspects is a different way of applying metrics to the refactoring process. The explicit use of software metrics to locate problem code for (non-AO) refactoring has been tried [54]. Cole and Borba propose what they call AspectJ laws, a catalogue of code transformations [55].

JHotDraw, a Java version of the HotDraw library mentioned in Section 2.2, has been used as a test-bed for AOSD work. Note that HotDraw is similar to GEF in design, complexity and function. AJHotDraw is an open source AO reengineered version of JHotDraw created to test the feasibility of reengineering legacy code with aspects. Ceccato et al. used JHotDraw to compare aspect mining techniques [56]. A different development process from ours was used, a four step process consisting of mining, exploration, documentation, and refactoring based on so-called crosscutting concern sorts. Hannemann et al. show the viability of a role-based approach to semi-automate AO refactoring by refactoring three different design patterns - observer, singleton and template method – also in JHotDraw [57].

6 Conclusions and Future Directions

Having analyzed the empirical results and reviewed existing research in the area of aspect-oriented reengineering it is clear there is potential in the areas of reducing complexity, maintainability and promoting reuse. There are varying degrees of success depending on the extensiveness of the reengineering and the type of system it is being applied to. The results we obtained from applying a suitable metric suite to both the original library and the reengineered library suggest that the introduction of aspects did show slight improvements in many fundamental measures of software quality in our short iteration approach. The key question is if this improvement warranted the effort. Future work is needed on defining benefit in terms that factor in development costs. Extensive AO re-design may be difficult within or incompatible with the short iterations in the most common agile processes. We conclude that without greater automation in the form of tools and a supportive process, AO reengineering of working OO software in an agile process is hard to justify.

Constraints and limitations of this study where discussed in Section 5. Future work needs to look at issues surrounding the practical application of AO refactoring in agile development including team development, training, tool support, testing, and quality control. Beuche and Beushe highlighted major issues with transferring aspect technology into practice [58] that can serve as a guide to needed work in the area. They state that AO programming has yet to prove its value in terms of making software development cheaper and that AO programming might be useful for certain functions but not all. Ascertaining how AO refactoring can be most judiciously employed and incorporated into existing processes is an important factor. It is also worth noting that AO programming is still little used outside the Java community and large-scale success stories are few; but there are island of success, see [47, 59, 23] for the state-of-the-art in large-scale deployment. For large code bases it can be difficult to balance the amount of time spent investigating areas where AO can be introduced, and the overall benefit gained from doing so. In such cases prior developer knowledge of the system being reengineered could be advantageous to tip the balance in favour of AO refactoring as well as use of the automation tools discussed in Section 5.

Acknowledgments

I would like to thank the MSc student Mark Donnelly who worked with me on the AspectJ coding.

References

[1] Dyba, T., Dingsoyr, T. 2009. What do we know about agile software development? IEEE Software , 26(5), pp.6-9.

[2] Parnas, D. 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(1).

[3] Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M., 1999. N degrees of separation: multi- dimensional separation of concerns, in: International Conference on Software Engineering, IEEE Press, New York, NY, pp. 107-119.

[4] Elrad, T., Filman, R.E., Bader, A., 2001. Aspect-oriented programming introduction. Communications of the ACM, 44(10), 2001.

[5] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J., 1997. Aspect-oriented programming, in: Aksit, M., Matsuoka, S. (Eds.), ECOOP 1997: LNCS, vol. 1241, Springer, Heidelberg, pp. 220 – 242.

[6] Katz, S., Mezini, M., Kienzle J., (Eds.), 2010. Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling. Lecture Notes in Computer Science 6210, Springer, Heidelberg.

[7] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., Lucena, C., 2006. Quantifying the effects of aspect-oriented programming: a maintenance study, in: Proc. IEEE International Conference on Software Maintenance, pp. 223-233.

[8] Araújo, J., Baniassad, E.L.A., 2007:.Guest editors' introduction: early aspects - analysis, visualization, conflicts and composition. T. Aspect-Oriented Software Development 3: 1-3

[9] Fanta, R., Rajlich, V., 1999. Restructuring legacy C code into C++. in: International Conference on Software Maintenance, IEEE Press, New York, NY, pp. 77-85.

[10] Ladded, R., 2003. AspectJ in Action, Manning, Greenwich, CT.

[11] Kiczales, G., 2004. The AOP report card, Software Development, January, CMP Media.

[12] Colyer, A., Clement, A., Harley, G., Webster, M., 2004. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison-Wesley Professional,.

[13] LaToza, T.D., Venolia, G., DeLine, R., 2006. Maintaining mental models: a study of developer work habits, in Proceedings of the 28th international Conference on Software Engineering (ICSE '06), ACM Press, New York, NY.

[14] Chikofsky, E., Cross, J., 1990. Reverse engineering and design recovery: A taxonomy, IEEE Software, 7(1), pp. 13-18.

[15] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, Boston, MA.

[16] Martin, R.C., 2002. Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, Upper Saddle River, NJ.

[17] Beck, K., 2000. Extreme Programming Explained: Embrace Change, Addison Wesley.

[18] Lippert, M., Roock, S., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, Wiley.

[19] Kendall, E.A., 2000. Reengineering for separation of concerns, in: Tarr, P., Finkelstein, A., Harrison, W., Nuseibeh, B., Ossher, H., Perry, D. (Eds.), Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE 2000.

[20] Chidamber, S.R. Kemerer, C.F. 1996. A metric suite for object-oriented design, IEEE Transactions on Software Engineering, 20(6), pp. 476–493.

[21] Martin, R.C., 1994. OO design quality metrics: an analysis of dependencies, in: Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA 1994, ACM Press, New York, NY.

[22] Weinberg, G., 1997. Quality Software Management: Anticipating Change, 4, Dorset House, New York, pp. 13-20.

[23] Rashid, A., Cottenier, T., Greenwood, P., Chitchyan, R., Meunier, R., Coelho, R., Sudholt, M., Joosen, W., 2010. Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe, IEEE Computer, 43(2), pp.19-26.

[24] Rosenberg, L.H., Hyatt, L.E., 1997. Hybrid re-engineering, in: Third IEEE International

Symposium on Requirements Engineering (ISRE), IEEE Press, New York, NY.
[25] Kataoka, Y. Imai, T. Andou, and H. Fukaya, T., 2002. A quantitative evaluation of maintainability enhancement by refactoring, in: Proc. International Conference on Software Maintenance.
[26] Pizka. M., 2004. Straightening Spaghetti Code with Refactoring, in: Proc. of the Int. Conf. on Software Engineering Research and Practice - SERP, CSREA Press, pp 846- 852.
[27] Monteiro, M.P., Fernandez, J.M., 2006. Towards a catalogue of refactorings and code smells for AspectJ, in: A. Rashid and M. Aksit (Eds.), Transactions of Aspect-Oriented Software Development I: LNCS 3880, Springer, Heidelberg, pp. 214 – 258.
[28] Gamma, E,. Helm, R., Johnson, and R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional.
[29] Denier, S., Comte, P., 2006. Understanding design pattern density with aspects, Software Composition 5th international symposium.
[30] Rashid, A., Sawyer, P., 2001. Aspect-oriented and database systems: an effective customization approach, IEEE Software 148(5), pp. 156-164,
[31] Hanenberg, S., Oberschulte, and C., Unland, R., 2003. Refactoring of aspect-oriented software, in Unland, R. (Ed.), Lecture Notes in Computer Science, volume 2591, Springer, Heidelberg, 2003.
[32] Laddad, R., 2003. Aspect-oriented refactoring, Parts 1 and 2, The Server Side, http://www.theserverside.com/tt/articles/article.tss?l =AspectOrientedRefactoringPart1
[33] Demeyer, S., Ducasse, S., Nierstrasz,O., 2002. Object-Oriented Reengineering Patterns, Morgan Kaufmann, 2002.
[34] Binkley, D., Ceccato, M., Harman, M., Ricca, and F., Tonella, P., 2005. Automated refactoring of object-oriented code into aspects, in: 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE Press, New York, NY, pp. 27–36.
[35] Apel, S., 2010. How AspectJ is Used: An Analysis of Eleven AspectJ Programs, Jounal of Object Technology (JOT), 9(1), pp. 117-142.
[36] Henderson-Sellers, B., 1994. Book Two of Object-Oriented Knowledge, Prentice Hall, Upper Saddle River, NJ.
[37] Hannemann, J., Kiczales, G., 2002. Design pattern implementation in Java and AspectJ, in: OOPSLA ‘02, ACM Press, New York, NY.
[38] Zakaria, A.A., Hosny, H., 2003. Metrics for aspect-oriented software design, in: Third International Workshop on Aspect Oriented Modeling at International Conference on Aspect-Oriented Software Development, ACM Press, New York, NY.
[39] Wilking, D., Khan, U.F., Kowalewski, S., 2007. An empirical evaluation of refactoring, e-Informatica Software Engineering Journal, 1(1).
[40] Bourqun, F., and Keller, R.K., 2007. High-impact refactoring based on architecture violations, in: Conference on Software Maintenance and Reengineering - CSMR , pp. 149-158.
[41] Thomas, W., Delis, A., Basili, V.R., 1997. An analysis of error in a reuse-oriented development Environment. Journal of Systems and Software, 38(3), 1997.
[42] Chen, C.-T., Cheng, Y.C., Hsieh, C.Y., Wu, I.-L., 2009. Exception handling refactorings: Directed by goals and driven by bug fixing, Journal of Systems and Software, 82(2), pp. 333-345.
[43] Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A., Kersten, M.A., Does aspect-oriented programming work? Communications of the ACM, 44(10), pp. 75-77.
[44] Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant’Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A., 2007. On the impact of aspectual decompositions on design stability: an empirical study, in: Ernst, E. (Ed.), ECOOP 2007: LNCS, vol. 4609, Springer, Heidelberg, pp. 176 - 200.
[45] Kellens, A., Mens, K., Tanella, P., 2007. Survey of automated code-level aspect mining Techniques, in: Rashid, A., Aksit, M. (Eds.), AOSD IV: LNCS 460, Springer, Heidelberg, pp. 14-162.
[46] Binkley, D., Ceccato, M., Harman, M., Tonella, P., 2006. Tool supported refactoring of existing object-oriented code into aspects, IEEE Transactions on Software Engineering.
[47] Colyer, A., Clement, A., 2004. Large-scale AOSD for middleware. in: Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD '04). ACM Press, New York, NY, pp. 56-65.
[48] Filho, F.C., Rubira, C.M., Maranhão Ferreira, R., Garcia, A., 2006. Aspectization of exception handling: A quantitative study, in:Advanced Topics in Exception Handling Techniques: LNCS vol. 4119, Springer, Heidelberg, pp. 255-274.
[49] Lippert, M., V. Lopes, C., 2000. A study on exception detection and handling using aspect-oriented programming, in: International Conference Software Engineering (ICSE 2000), ACM Press, New York, NY, pp. 418-427.
[50] Tsang, S.L., Clarke, S., Baniassad, E., 2004. An evaluation of aspect-oriented programming for Java-based real-time systems development, in: 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Press, New York, NY, pp. 291-300.
[51] Sullivan, K.J., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H., 2005. Information hiding interfaces for aspect-oriented designs, in: 10th European Software Engineering Conference, ACM Press, New York, NY, pp. 166–175.
[52] Zhang, C., Jacobsen, H., 2004. Resolving feature convolution in middleware systems, SIGPLAN

Notices, 39(10), ACM Press, New York, NY, pp. 188–205.

[53] Madeyski, L., Szala, L., 2007. Impact of aspect-oriented programming on software development and design quality, IET Software, 1(5), pp. 180-187

[54] Simon, F., Steinbreuckner, F.C., Lewerentz, C., 2001. Metrics based refactoring. in: European Conference on Software Maintenance and Reengineering, pp. 30-38.

[55] Cole, L. Borba, P., 2005. Deriving refactorings for AspectJ, in: AOSD IV, ACM Press, New York, NY, pp. 123-134.

[56] Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T., 2005. A qualitative comparison of three aspect mining techniques, in: 13th International Workshop on Program Comprehension, IEEE Press, New York, NY, pp. 13–22.

[57] Hannemann, J., Murphy, G., Kiczales, G., 2005. Role-based refactoring of crosscutting concerns, in: 4th International Conference on Aspect-Oriented Software Development, ACM Press, New York, NY, pp. 135 -146.

[58] Beuche, D., Beust, C., in: Colyer, A.M. , Kawakami Harrop Galvão, R., Johnson, R., Vasseur, A., Beuche, D., Beust, C., (Eds.) 2006. Point/counterpoint, IEEE Software 23(1), pp. 72-75.

[59] Wiese, D., Meunier, R., 2008. Large-scale application of AOP in the healthcare domain: A case study, in: 7th AOSD, ACM Press, New York, NY.

Web References

1. http://www.eclipse.org/ajdt/
2. http://gef.tigris.org/
3. http://argouml.tigris.org/
4. http://aopmetrics.tigris.org/