Dependencies in the Context of Aspect-Oriented Programming

by

Stephan Gudmundson

B.Sc., University of British Columbia, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in THE FACULTY OF GRADUATE STUDIES (Department of Computer Science)

We accept this thesis as conforming to the required standard

The University of British Columbia

May 2002

© Stephan Gudmundson, 2002
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 22, 2002
Abstract

Software developers use dependencies to understand designs. An inappropriate dependency can be confusing, reduce the flexibility of the system or re-usability of a module, or cause problems in system maintenance or evolution. This thesis proposes three new kinds of dependencies: whole-class, whole-subsystem and call-graph dependencies. Several examples of each dependency are encountered in the analysis of two implementations of the same system, one in the AspectJ programming language and the other in the Java programming language. The expected effect of these dependencies on comprehensibility, evolvability and re-usability is described. Finally, the new dependencies proposed in this thesis are set in the context of existing work.
# Contents

Abstract ii

Contents iii

List of Tables v

List of Figures vi

1 Introduction 1

2 Dependency Characterization 2

2.1 Overview ........................................ 2

2.2 Name-based Dependencies .......................... 3

2.3 Whole-Class Dependencies .......................... 4

2.3.1 Whole-Interface Dependencies ................. 4

2.3.2 Whole-Implementation Dependencies .......... 5

2.4 Whole-Subsystem Dependencies ................... 5

2.5 Call Graph Dependencies .......................... 6

3 Dependency-Based System Analysis 7

3.1 Overview ......................................... 7

3.2 The System ...................................... 7

3.3 Concerns ........................................ 8
List of Tables

3.1 Dependency differences in the implementation of point bounding ........ 13
3.2 Correspondence between dependency differences in point bounding .... 14
3.3 Dependency differences in the implementation of basic move notification ... 15
3.4 Correspondence between dependency differences in basic move notification .. 17
3.5 Dependency differences in the implementation of nested move notification suppression ........................................ 18
3.6 Correspondence between dependency differences in nested move notification suppression ........................................ 18
3.7 Dependency differences in the implementation of display updating ........ 18
3.8 Correspondence between dependency differences in display updating ...... 19
3.9 Dependency differences in the implementation of tracing .................. 20
3.10 Correspondence between dependency differences in tracing ............... 20
3.11 Dependency differences in the implementation of factory pattern enforcement 21
3.12 Correspondence between dependency differences in factory pattern enforcement 22
3.13 Dependency Difference Summary ........................................ 23
3.14 Modularization Difference Summary ..................................... 24
List of Figures

3.1 Dependency differences created by using explicit method invocations 24
3.2 Using an abstract method to obscure the true identity of the invocation target 25
3.3 Whole-class dependency addressed with explicit method invocations 26
3.4 Whole-class dependency addressed with pointcut and advice 26
Chapter 1

Introduction

Software developers use dependencies to understand designs. A dependency is a connection between implementation entities that indicates that one of the entities can affect the other. For example, a dependency is created whenever a software module invokes functionality in another software module. Inappropriate dependencies can be confusing, reduce the flexibility of the system or the re-usability of a module, or cause problems in the maintenance and evolution stages[7].

In this thesis, I propose three new kinds of dependencies for software developers to consider. These dependencies have come to light in the context of aspect-oriented programming[3], but they are not introduced by AOP. To demonstrate this, we provide examples in both an AOP and non-AOP programming language. This thesis uses the AspectJ[4] and Java™ programming languages and assumes that the reader is familiar with their syntax and semantics.

The remainder of this thesis is organized as follows. Chapter 2 describes the dependencies generally. Chapter 3 presents an application of the use of the dependencies in analyzing implementations of a simple system in AspectJ and Java. Chapter 4 discusses some of the implications of the results of the analysis in Chapter 3. Chapter 5 surveys related work and Chapter 6 concludes the thesis.
Chapter 2

Dependency Characterization

2.1 Overview

This thesis proposes three new kinds of dependencies and positions the existing common notion of dependency relative to these. It is not the intent of this thesis to develop a comprehensive set of dependency types, but only to identify several kinds of dependencies within the general category of dependencies.

The dependencies used in this thesis are described briefly below.

1. Name-based dependencies. These are the dependencies created wherever some element is referenced by name. In Java and AspectJ, the referent can be a class, interface, aspect, method, field, or pointcut. For example, a method invocation is a use of the name of the invoked method. These are the dependencies that developers have used for decades to understand system structure.

2. Whole-class dependencies. These are dependencies created wherever the whole class depends on something, or something depends on the whole class. For instance, an aspect might need to know all of the ways that a Point can be moved, so that it can react to movements consistently across all move operations.

3. Whole subsystem dependencies. These are dependencies created wherever some element must know all of the classes in the system or a specific subsystem (package,
packages or entire system) that meet some criteria.

4. **Call graph dependencies.** These are dependencies created wherever some element makes assumptions about the nature of the call graph. For example, in AspectJ, such dependencies typically arise from using \textit{cflow} pointcuts.

### 2.2 Name-based Dependencies

These dependencies arise when one syntactic element names another. The common notion of dependency is the name-based dependency. In Java, there are several ways that name-based dependencies are created:

- A field access introduces a dependency on the field signature onto the code making the access. For example, if method \texttt{getColor} accesses the variable \texttt{color}, then \texttt{getColor} depends on \texttt{color}.

- A method invocation introduces a dependency on the method signature onto the code making the invocation. For example, if method \texttt{setWidth} calls method \texttt{setX}, then \texttt{setWidth} depends on \texttt{setX}.

- A field, parameter, or local variable declaration introduces a dependency on the type of the declaration onto the code making the declaration. For example, the field declaration \texttt{Color color} depends on the class \texttt{Color}.

- A method signature depends on its return type. For example, the method \texttt{Color getColor()} depends on class \texttt{Color} because its return value is of type \texttt{Color}.

- Expressions depend on all types of nested expressions. For example, \texttt{rect.getColor().toString()}, where \texttt{rect} is of type \texttt{Rectangle}, depends on \texttt{Rectangle}, \texttt{Color}, and \texttt{String} (the return type of the \texttt{toString} method). Even though the \texttt{Color} class is not named explicitly, the expression depends on it through the method signature \texttt{Color getColor()}.

- Inheritance and interface implementation introduce a dependency on the type of the extended or implemented onto the extending or implementing type. For example, in
the declaration class Rectangle implements Shape, the Rectangle class depends on the Shape interface.

- Method overriding or implementation of an abstract/interface method introduce a dependency on the signature of the overridden/implemented method onto the overriding/implementing method. For example, if setWidth in class Rectangle is an implementation of the setWidth method declared by interface Shape, then Rectangle.setWidth depends on Shape.setWidth.

AspectJ's additional language features can introduce name-based dependencies as well. Pointcuts can name methods, fields, class, interfaces and aspects, and in so doing become dependent on the referent. Advice depends on the pointcut at which it applies. Aspects can depend on classes or interfaces in their extends, implements, and of each clauses.

2.3 Whole-Class Dependencies

Whole-class dependencies arise when either an entity depends on the whole class, or the whole class depends on an entity. Whole class does not mean that the dependency affects or is affected by literally every element in the class, but rather that the dependency is not confined to any particular element or elements of the class. We identify two kinds of whole-class dependencies: whole-interface and whole-implementation dependencies. The only difference between a whole-implementation and whole-interface dependency is the level of knowledge that the dependent entity requires. A whole-interface dependency indicates that the dependent needs only information about the interface, and not the implementation, of the module. A whole-implementation dependency indicates that the dependent needs information about both the interface and implementation of the module.

2.3.1 Whole-Interface Dependencies

These are dependencies upon the interface of a class. For example, consider the pointcut signature pointMoves(Point pt). This pointcut signature implies that the pointcut promises
to identify every join point at which a Point moves. This is a dependency on the whole interface of Point.

A whole-interface dependency is different from a name-based dependency on a class, such as would be created by a variable declaration that names the class. In that case, the dependency is only upon the name of the class, while the whole-interface dependency targets the interface elements (methods and/or fields) contained within the class.

A whole-interface dependency is also different from the name-based dependencies on particular elements of a class created by a method call site. For a method call, the client only needs to know that the identified method is one way to get a given task done. In the whole-interface case, the client needs to know all of the ways that a task might be done. The difference is clear in the context of evolution. For instance, consider a client X wishing to move a Point instance pt dx units to the right. X can use pt.setX(pt.getX()-dx) to implement this operation. As an evolutionary change, suppose the Point class receives a moveBy method. X is not obligated to use this new interface, and is therefore not affected by its addition. However, the pointMove pointcut must be updated to capture calls to the new moveBy method.

2.3.2 Whole-Implementation Dependencies

These are dependencies on the whole implementation of a class. For example, suppose that Points are to be bounded within a given region of the plane. One implementation of this would have Point methods that alter coordinate values check that the new coordinates are valid before making any changes. The whole-implementation dependency in this case is that the whole implementation must perform these checks.

2.4 Whole-Subsystem Dependencies

These are dependencies upon the whole collection of classes, interfaces and aspects in a given package, packages or the entire system. For example, consider the pointcut move(): Rectangle.move() || Ellipse.move() (where the move pointcuts in Rectangle and Ellipse capture
movements of Rectangle and Ellipse). This pointcut is intended to capture the movements of all shapes, so it needs to know all of the classes that represent shapes. This is a whole-subsystem dependency.

As with whole-class dependencies, the term whole-subsystem dependency does not imply that the dependent is tied to everything in the subsystem, but rather that the dependency cannot be localized to particular modules in the subsystem.

### 2.5 Call Graph Dependencies

These are dependencies upon the dynamic call graph of the system. These dependencies arise commonly in using AspectJ's cflow construct. For example, the pointcut `call(void Shape.repaint()) && cflow(call(void Shape.setColor(Color))` captures direct or indirect calls from Shape's `setColor` method to Shape's `repaint` method.

Call-graph dependencies usually include an indication of the purpose of the call. For instance, for the pointcut `colorChangeRepaint():call(void Shape.repaint()) && cflow(call(void Shape.setColor(Color))` indicates not only that `setColor` may call `repaint`, but that the purpose of the call is to repaint the display because the color has changed.

Call-graph dependencies exist in Java as well. For instance, the Swing library in JDK 1.2+ depends on calls to a variety of Swing methods being made in a certain Swing dispatch thread. This dependency allows Swing to avoid some locking and consequently improve the library's performance.
Chapter 3

Dependency-Based System Analysis

3.1 Overview

This chapter uses the dependencies introduced in the preceding chapter to reason about design. We consider two implementations of similar functionality, one in AspectJ and the other in Java. This analysis has two purposes. First, it shows the use of the dependencies in the context of both an AOP and non-AOP language, which demonstrates that the dependencies are not specific to AOP. Second, it provides a basis for comparing the AOP and non-AOP implementations, which provides some insight into the new capabilities offered by AOP.

3.2 The System

This example is taken from the “figures” example used in [5]. I obtained the AspectJ implementation from the AspectJ group\(^1\) and derived an almost-equivalent-functionality pure Java implementation.

\(^1\)from http://www.aspectj.org
The basic functionality is as follows. The system is concerned with representing and rendering 2D figures. The FigureElement interface represents a 2D object; in this implementation, only Point and Line are provided, but others like Rectangle, Ellipse, etc are easily imagined.

FigureElements are components of a Figure, which is the entire 2D entity. Figure acts as a factory for its FigureElement components; the programmer is not to construct FigureElements directly.

A Figure is rendered onto a Display, though this functionality is not provided in this sketch of the implementation. The only interaction with Display is: whenever a Figure changes, Display's needsRepaint method must be called to update the visual image presented to the user.

For the Java and AspectJ implementations described below, there are several alternatives for the implementation. I have chosen what I believe to be representative of implementations one would see in practice. I have considered several other implementations, and the conclusions drawn in this chapter hold for most of these other implementations as well.

3.3 Concerns

The analysis is driven by examining specific concerns in the system. We consider three types of concerns: classes, rules, and features. A class is the usual notion of state plus behavior. A rule is a requirement of the system. A feature is a behavior that is not localized in the class structure. In our example, we will consider the following concerns:

1. The Point, Line, Figure, and Display classes and the FigureElement interface. These are the primary data abstractions in the system.

2. Point's x- and y-coordinates are to be restricted to the range 0-100. This feature is referred to as point bounding.

3. Whenever any FigureElement moves, interested parties should be informed. This feature is referred to as (basic) movement notification.
3.1. As an extension to basic move notification, an additional requirement to suppress move notifications for logically nested FigureElements (such as the two Points that make up a Line). This rule is referred to as nested move notification suppression.

4. Several messages are to be printed to the console for debug purposes. This feature is referred to as tracing.

5. The display must be updated whenever the visual appearance of a FigureElement changes. This rule is referred to as display updating.

6. The system architecture mandates that programmers obey the factory pattern established by the Figure class for creating FigureElement object instances. This rule is referred to as factory pattern enforcement.

3.4 AspectJ Implementation

The AspectJ and Java implementations are largely similar. This subsection describes the AspectJ-specific parts of the implementation. It makes use of 4 aspects: Tracing, PointBoundsChecking, DisplayUpdating, and FactoryEnforcement.

Tracing prints messages whenever Point.setX/Y or Display.needsRepaint is called. In the case of Display.needsRepaint, the tracing message includes an indication of whether the call was made directly by DisplayUpdating or not. The tracing messages are produced by advice on calls to the methods.

The PointBoundsChecking aspect ensures that the coordinates of a Point remain within the range of 0-100. This is achieved with advice that runs before the Point.setX/Y methods. The advice throws an error if the parameter to the set call would move the Point outside of the valid area.

The FactoryEnforcement aspect provides a compile-time check that only the factory methods in Figure construct Point and Line. That is, it ensures that there are no calls to constructors of Point or Line outside of the factory methods in Figure. It has no effect on the behavior of the system.
DisplayUpdating implements the requirement that Display.needsRepaint be called whenever the appearance of a Figure changes. This is implemented by advising calls to methods that can change the appearance of FigureElements, which in this case are only the operations that move FigureElements. The detailed semantics are that DisplayUpdating does not know which specific FigureElement has changed, and that nested move operations do not trigger repaints. The only nested move operation is within Line.moveBy, which nudges a line by given X and Y deltas. This method calls down into Point.moveBy. DisplayUpdating ensures that such calls to Point.moveBy do not trigger calls to Display.needsRepaint by using cflow to eliminate these nested calls in the pointcut definition.

3.5 Java Implementation

This subsection describes how the AspectJ-specific parts of the implementation described above are implemented in Java.

Tracing is implemented by scattering the print statements throughout the code base. For instance, the bodies of Point._setX/Y include calls to Trace.println. (The difference between .setX and setX is described below.) Noting whether or not the call to Display.needsRepaint came from DisplayUpdating is not easily implemented in Java, so the behavior is slightly different: all calls to Display.needsRepaint log the fact that the method was called, and all call sites within DisplayUpdating log the fact that they are about to call Display.needsRepaint. This provides the equivalent information, but in a slightly different form: the presence or absence of the line from DisplayUpdating indicates whether or not the call to Display.needsRepaint came from DisplayUpdating. I believe that this is the most reasonable Java implementation, even if it deviates slightly from the AspectJ behavior.

Similarly, bounds checking for Point is implemented by moving the tests for boundary violations into the Point._setX/Y methods.

The compile-time FactoryEnforcement aspect can not be implemented statically as it is in AspectJ, because the standard Java compiler does not support this kind of static checking. As a compromise, the check is implemented at run-time instead. A special Token
class is introduced, which can only be created by the factory using Java's scoping rules. The constructors for Point, Line, and any other FigureElements take this type as a parameter in their constructors. If the token is non-null, then it must have been created by the factory. As long as the token is only given to trusted code, then it is proof that the factory is creating the FigureElement. If the token is null, then an exception is thrown to indicate the "unauthorized" attempt to construct the FigureElement.

Implementing DisplayUpdating's functionality in the Java model is more complex. The straightforward approach is to simply place calls to Display.needsRepaint directly into Point, Line and any other FigureElements that are added. However, this solution ties the FigureElement classes too closely to the display updating concern for most Java programmers.

Instead, the familiar JavaBeans™ event model is employed. The concept of "move events" is added at the FigureElement interface level. This has the effect of requiring all FigureElement classes to support the event model. Whenever a FigureElement moves, it must notify any registered MoveListeners by dispatching a MoveEvent. The event includes the identity of the moved FigureElement, but not the "before" and "after" positions.

It is clear that the event machinery is doing more work than the DisplayUpdating.move pointcut. The event mechanism is an interface for use by a wide range of clients, whereas the move pointcut is only for the DisplayUpdating aspect. As we will see, this difference has an impact on modularity, but we will be careful to point out where the modularity differences arise because of this difference in functionality.

To relieve Point and Line of some burden, the listener registration methods and event dispatching functionality are implemented in the AbstractFigureElement class. Point and Line inherit from this class. They must call the protected AbstractFigureElement.fireMoveEvent method whenever they move.

As mentioned earlier, the DisplayUpdating aspect in the AspectJ implementation avoids capturing nested move operations in its advice. I decided that this was an important

\[^2\text{http://java.sun.com/products/javabeans/}\]
behavioral property that must be maintained in the Java version. Supporting this behavior requires several changes to Point and Line.

Consider Point first. The implementation of Point.moveBy calls setX and setY to effect the move. If setX and setY were to call fireMoveEvent, then the call to Point.moveBy would generate at least two move events, one each for setX and setY, instead of one. To avoid this problem, Point is instead structured such that setX and setY are called only by clients, and two new methods, _setX and _setY, are called internally. The internal methods do not fire move events.

This pattern can suppress generation of nested events within a single class, but it does not solve the problem of nested operations across classes. In particular, Line.moveBy call Point.moveBy on each of its endpoints to effect the move. This would generate at least two move events, one from each endpoint, which is not acceptable.

To solve this problem, Point.moveBy must be informed that event generation should be suppressed in some situations. This is accomplished by introducing an overloaded form of Point.moveBy that includes a boolean parameter fireEvent which indicates whether or not the move operation should generate an event.

This pattern of overloading each move method with a form that includes a specific directive regarding event generation is repeated for all move methods on Point and Line. However, this is not propagating up to the FigureElement interface, because “regular” clients of Point and Line should not have control over event generation. In effect, it is a special interface.

It would have also been possible to solve this problem using thread-local storage. In this design, the fireEvent indicator is stored in a thread-local variable instead of being passed on the stack. The two approaches are similar enough that we will only consider the fireEvent parameter version.
## 3.6 Dependency Differences

### 3.6.1 Point Bounding Feature

In both implementations, the point-bounding feature is implemented in two parts. The check of whether a proposed coordinate falls within the valid range is implemented by `checkX` and `checkY`. In order to implement the bounding consistently, the implementation must also include the rule that "all potential moves of Points are validated before being applied."

In the AspectJ version, the `PointBounding` aspect implements the point-bounding feature. The `checkX` and `checkY` methods are in the aspect and the "moves must be validated" rule is implemented by the before advice.

In the Java version, the `Point` class implements the point-bounding feature. The `checkX` and `checkY` methods are in this class. The "moves must be validated" results in a non-localized dependency throughout the `Point` class.

The dependency differences between the two implementations of this concern are shown in Table 3.6.1 and correlated in Table 3.2.

<table>
<thead>
<tr>
<th>AspectJ Implementation</th>
<th>Java Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Point</strong></td>
<td><strong>Point</strong></td>
</tr>
<tr>
<td>- setX</td>
<td>(whole i/f)</td>
</tr>
<tr>
<td>- setY</td>
<td></td>
</tr>
<tr>
<td><strong>PointBoundsChecking</strong></td>
<td></td>
</tr>
<tr>
<td>- before Point.setX</td>
<td></td>
</tr>
<tr>
<td>- before Point.setY</td>
<td></td>
</tr>
<tr>
<td><strong>Core Point Class</strong></td>
<td></td>
</tr>
<tr>
<td>- setX</td>
<td></td>
</tr>
<tr>
<td>- setY</td>
<td></td>
</tr>
<tr>
<td><strong>Point Bounds Checking</strong></td>
<td>checkX</td>
</tr>
<tr>
<td></td>
<td>checkY</td>
</tr>
<tr>
<td><strong>Rule:</strong> all methods must check new coordinates</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Dependency differences in the implementation of point bounding

In the AspectJ implementation, the pointcut used by the before advice in the `PointBoundsChecking` aspect names `Point.setX`, `setY` and `moveBy`. In addition, the aspect knows all the ways to change the coordinates of a `Point`. 
In the Java implementation, the core Point class depends on the point bounding concern. Specifically, all of the methods that change a Point's coordinates call the checkX and/or checkY methods. In addition, the entire implementation of the core Point class depends on the “must validate coordinates” rule. This is a whole-implementation dependency because adding any new means of changing a Point's coordinate would be required to follow this rule.

Table 3.2: Correspondence between dependency differences in point bounding

<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>before Point.setX → Point.setX; before Point.setX → checkX</td>
<td>Point.setX → Point.checkX</td>
<td>Reversed</td>
</tr>
<tr>
<td>PointBoundsChecking → Whole interface of Point</td>
<td>Whole implementation of Point → “must validate coords” rule</td>
<td>Reversed</td>
</tr>
</tbody>
</table>

3.6.2 Move Notification Feature

We will consider basic move notification, and then suppression of notification of nested moves. Basic move notification would be without the suppression of nested move notifications. In the Java implementation, all of the methods with a fireEvent parameter would be absent, and all move methods would unconditionally call fireMoveEvent as their last action. In the AspectJ implementation, the cflow component would be absent from the advice.

Basic Move Notification

In the AspectJ implementation, this concern is contained completely within the DisplayUpdating.move pointcut. Clients may receive movement notifications by attaching advice to this pointcut.

In the Java implementation, the movement notification feature is implemented in several components. AbstractFigureElement provides the implementation of the event dispatching functionality. In addition, this design introduces a rule: “all FigureElement realizations must initiate movement notification whenever they are moved.”
The dependency differences between the two implementations of this concern are shown in Table 3.6.2 and correlated in Table 3.4.

<table>
<thead>
<tr>
<th>AspectJ Implementation</th>
<th>Java Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>All classes that implement FigureElement</td>
<td>FigureElement</td>
</tr>
<tr>
<td>DisplayUpdating - pointcut move</td>
<td>FigureElement Core - moveBy</td>
</tr>
<tr>
<td>(whole subsystem)</td>
<td>Move Notification - addMoveListener</td>
</tr>
<tr>
<td>FigureElement - moveBy</td>
<td>AbstractFigureElement - addMoveListener - fireMoveEvent</td>
</tr>
<tr>
<td>(whole i/f)</td>
<td>Point - setX</td>
</tr>
<tr>
<td>Point - setP1</td>
<td>Line - setP1</td>
</tr>
<tr>
<td>(whole i/f)</td>
<td>Rule: move methods must initiate notification.</td>
</tr>
<tr>
<td>Line - setX</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Dependency differences in the implementation of basic move notification

In the AspectJ version, the DisplayUpdating.move pointcut names specific methods in FigureElement, Point and Line. In addition, DisplayUpdating must know all of the ways that each kind of FigureElement can move (the whole interface dependencies). DisplayUpdating also needs to know all of the classes that implement FigureElement (the whole-subsystem dependency). Finally, Point and Line refer to the FigureElement interface in their implements clause.

In the Java version, FigureElement contains the move notification concern. Ab-
AbstractFigureElement names the FigureElement interface in its implements clause, and implicitly names FigureElement.addMoveListener (the dependency is introduced because the signature in AbstractFigureElement must match the signature in FigureElement exactly, and so the implementers are tied to the abstract declaration). Each FigureElement class names AbstractFigureElement as its parent. Each of the methods in classes that implement FigureElement call AbstractFigureElement.fireMoveEvent.

The Java version has another collection of dependencies relating to the “move methods must initiate notification” rule. This rule exists at the subsystem level, rather than being localized within any particular class. All realizations of FigureElement must obey this rule throughout their implementation.

Nested Move Suppression Feature

The dependency differences between the two implementors of this concern are shown in Table 3.6.2 and correlated in Table 3.6.

In the AspectJ implementation, the pointcut to which the after advice applies is based on an assumption about the call graph, as is described above.

In the Java implementation, the implementations of Point and Line depend on the nested notification suppression rule. In any situation, before initiating a move notification, one would have to ensure that the move operation is not logically nested within some larger-scope move operation.

The dependencies correspond as follows:

3.6.3 Display Updating Rule

The dependency differences between the two implementations of this concern are shown in Table 3.6.3 and correlated in Table 3.6.3.

In the AspectJ implementation, the DisplayUpdating aspect contains the implementations of both the display updating rule and the move notification concern. The display updating rule is enforced by receiving move notifications by advice attached to the Dis-
<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptc move -&gt; Point.setX/Y</code></td>
<td><code>Point.setX/Y -&gt; fireMoveEvent</code></td>
<td>Reversed (fireMoveEvent corresponds with pointcut move, in that they are both the interface for announcing movements)</td>
</tr>
<tr>
<td><code>ptc move -&gt; Line.setP1/P2</code></td>
<td><code>Line.setP1/P2 -&gt; fireMoveEvent</code></td>
<td>Reversed</td>
</tr>
<tr>
<td><code>ptc move -&gt; FigElt.moveBy</code></td>
<td><code>Point.moveBy -&gt; fireMoveEvent; Line.moveBy -&gt; fireMoveEvent</code></td>
<td>Reversed; in the Java case, each realization of the abstract method contains the dependency</td>
</tr>
<tr>
<td><code>Point -&gt; FigureElement</code></td>
<td><code>Point -&gt; AbstractFigureElement</code></td>
<td>Java impl uses inheritance</td>
</tr>
<tr>
<td><code>Line -&gt; FigureElement</code></td>
<td><code>Line -&gt; AbstractFigureElement</code></td>
<td>Java impl uses inheritance</td>
</tr>
<tr>
<td><code>n/a</code></td>
<td><code>AbstractFigElt -&gt; FigureElement; AbstractFigElt -&gt; fireMoveEvent</code></td>
<td>Introduced due to interface implementation</td>
</tr>
<tr>
<td><code>ptc move -&gt; &quot;all methods that move Points&quot;</code></td>
<td><code>Point -&gt; Rule: &quot;all move methods must initiate move notification&quot;</code></td>
<td>Reversed</td>
</tr>
<tr>
<td><code>ptc move -&gt; &quot;all methods that move Lines&quot;</code></td>
<td><code>Line -&gt; Rule: &quot;all move methods must initiate move notification&quot;</code></td>
<td>Reversed</td>
</tr>
<tr>
<td><code>ptc move -&gt; &quot;all methods that move FigureElements&quot;</code></td>
<td><code>n/a</code></td>
<td>In Java version, this is incorporated into the above two</td>
</tr>
<tr>
<td><code>ptc move -&gt; &quot;all classes that implement FigureElement&quot;</code></td>
<td><code>n/a</code></td>
<td>The Java version avoids this dependency by tangling move notification into the FigureElement interface</td>
</tr>
</tbody>
</table>

Table 3.4: Correspondence between dependency differences in basic move notification
DisplayUpdating
- after move && !cflow(move)

(call graph)
move operation is in the cflow of another move operation ⇔ a nested object is being moved

Point
- setX(x)
- setX(x, fireEvent)

(whole impl)

Line
- setP1(p1)
- setP1(p1, fireEvent)

Rule: notifications of moves of nested objects must be suppressed

<table>
<thead>
<tr>
<th>AspectJ Implementation</th>
<th>Java Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Dependency differences in the implementation of nested move notification suppression

<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association of nested method invocations with nested move operations</td>
<td>Point, Line → Nested move notification suppression rule</td>
<td>Transformed</td>
</tr>
</tbody>
</table>

Table 3.6: Correspondence between dependency differences in nested move notification suppression
DisplayUpdating

Move Notification
- pointcut move

Core
DisplayUpdating
- after move

Aspectsj Implementation
Java Implementation

<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>after move → ptc move</td>
<td>DisplayUpdater.moved → MoveListener.moved</td>
<td>Note that this name dependency is not reversed: in both cases, the dependency runs from the client to the interface.</td>
</tr>
<tr>
<td>n/a</td>
<td>Figure → addMoveListener; FigureElement → MoveListener</td>
<td>The Java implementation is more general and therefore has more components and more dependencies.</td>
</tr>
<tr>
<td>n/a</td>
<td>Figure → DisplayUpdater</td>
<td>Figure needs to connect the concrete objects that interact via abstract interfaces.</td>
</tr>
</tbody>
</table>

Table 3.7: Dependency differences in the implementation of display updating

playUpdating.move pointcut.

In the Java implementation, Figure owns an instance of DisplayUpdater. This instance is registered as a listener with any FigureElement instances created by the move* factory methods via FigureElement.addMoveListener. DisplayUpdater names the MoveListener interface in its implements clause and provides a realization for the FigureListener.moved method. FigureElement's addMoveListener method names MoveListener as a parameter.

Table 3.8: Correspondence between dependency differences in display updating
3.6.4 Tracing

The dependency differences between the two implementations of this concern are shown in Table 3.9 and correlated in Table 3.10.

<table>
<thead>
<tr>
<th>Tracing</th>
<th>AspectJ Implementation</th>
<th>Java Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointcut setXY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>after setXY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>after Display.needsRepaint</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Display          |                        |                     |
| needsRepaint     |                        |                     |

| DisplayUpdater   |                        |                     |
| moved            |                        |                     |

| Trace            |                        |                     |
| println          |                        |                     |

| Point            |                        |                     |
| setX             |                        |                     |
| setY             |                        |                     |

| Display Tracing  |                        |                     |
| DisplayTracing   |                        |                     |
| needsRepaint     |                        |                     |

| D.U. Tracing     |                        |                     |

Table 3.9: Dependency differences in the implementation of tracing

In the AspectJ implementation, the Tracing aspect refers to several methods in other classes. Perhaps more notable is the absence of any whole-interface dependencies, which pointcuts have created in all examples up to this point. Here, the Tracing aspect is applied to specific methods, whereas in other cases pointcuts were interested in all methods with a given property.

In the Java implementation, Point, Display and DisplayUpdater all contain separate tracing concerns; tracing in general is not recognizable as a coherent concern. All methods
that do tracing name Trace.println.

<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptc setXY \rightarrow Point.setX, setY</td>
<td>Point.setX, setY \rightarrow</td>
<td>Reversed</td>
</tr>
<tr>
<td>Trace.println</td>
<td></td>
<td></td>
</tr>
<tr>
<td>after Display.needsRepaint</td>
<td>Display.needsRepaint</td>
<td>Reversed</td>
</tr>
<tr>
<td>Display.needsRepaint</td>
<td>Trace.println</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.10: Correspondence between dependency differences in tracing

### 3.6.5 Factory Pattern Rule

The dependency differences between the two implementations of this concern are shown in Table 3.11 and correlated in Table 3.12.

In the AspectJ implementation, the `illegalNewFigElt` pointcut names the `FigureElement` and `Figure` classes directly. In addition, it assumes that every method in `Figure` (or a subclass) whose name starts with “make” is a `FigureElement` factory method, and vice-versa. Note that no whole-subsystem dependency is present, because AspectJ can determine...
all classes that implement the *FigureElement* interface automatically.

The AspectJ implementation also includes a call-graph dependency arising from the use of the *withincode* pointcut designator. The AspectJ idiom `call(z) & & withincode(y)` means "y directly calls x". This is more narrow than `call(z) & & cflow(y)` which means "y directly or indirectly calls x". In our example, the dependency is that the *Figure.make* directly constructs the *FigureElement* instance.

In the Java implementation, the factory enforcement rule is not localized within any single class. Instead, the implementation of each *FigureElement* class depends on the rule. In addition, there are references in the signatures of the *Point* and *Line* constructors as well as in the bodies of the *make* factory methods in *Figure* to the *Token* class.

<table>
<thead>
<tr>
<th>AspectJ Dependency</th>
<th>Java Dependency</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptc illegalNewFigElt -&gt; Figure;</td>
<td>Figure.makePoint -&gt; Token; Figure.makeLine -&gt; Token</td>
<td>Reversed and summarized</td>
</tr>
<tr>
<td>ptc illegalNewFigElt -&gt; &quot;factory methods start with make&quot;;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptc illegalNewFigElt -&gt; FigureElement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptc illegalNewFigElt -&gt; call-graph dependency</td>
<td>Point whole impl -&gt; factory enforcement rule; Line whole impl -&gt; factory enforcement rule</td>
<td>Transformed</td>
</tr>
</tbody>
</table>

Table 3.12: Correspondence between dependency differences in factory pattern enforcement

### 3.7 Summary

This section summarizes the differences between the two implementations from several perspectives. Table 3.13 summarizes the dependency differences on a per-concern basis. Table 3.14 summarizes the differences between the modularizations of the two implementations. Finally, per-dependency-type summaries are provided in sections 3.7.1 to 3.7.4.
<table>
<thead>
<tr>
<th>Concern</th>
<th>Dependencies only in Java impl.</th>
<th>Dependencies only in AspectJ impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>follows bounding, factory enforcement, move notification rules calls move notification names AbstractFigElt follows rule &quot;move ops issue move notifications&quot; follows rule &quot;move ops validate coords&quot;</td>
<td>(none)</td>
</tr>
<tr>
<td>Line</td>
<td>contains factory enforcement calls move notification names AbstractFigElt follows rule &quot;move ops issue move notifications&quot;</td>
<td>(none)</td>
</tr>
<tr>
<td>FigureElement</td>
<td>contains move notification concern</td>
<td>(none)</td>
</tr>
<tr>
<td>Figure</td>
<td>names factory enforcement names DisplayUpdater</td>
<td>(none)</td>
</tr>
<tr>
<td>Display</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>DisplayUpating</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>FactoryEnforcement</td>
<td>(none)</td>
<td>whole-interface dependency on Figure: named make* ↔ factory method</td>
</tr>
<tr>
<td>PointBoundsChecking</td>
<td>(none)</td>
<td>names Point.setX/Y, Line.setP1/P2, FigureElement.moveBy whole-subsystem dependency on all FigureElements for each FigureElement class, whole-interface dependency on all move methods</td>
</tr>
<tr>
<td>Tracing</td>
<td>(none)</td>
<td>names various methods</td>
</tr>
<tr>
<td>AbstractFigElt</td>
<td>contains impl of move notification names interface of move notification</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Table 3.13: Dependency Difference Summary
<table>
<thead>
<tr>
<th>Concern</th>
<th>Java</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>Contains bounding, logging, factory enforcement</td>
<td>(modularized)</td>
</tr>
<tr>
<td>Line</td>
<td>Contains factory enforcement</td>
<td>(modularized)</td>
</tr>
<tr>
<td>FigureElement</td>
<td>Contains move notification</td>
<td>(modularized)</td>
</tr>
<tr>
<td>Figure</td>
<td>Contains factory, “wiring” for display updating</td>
<td>Contains factory</td>
</tr>
<tr>
<td>Display</td>
<td>(modularized)</td>
<td>(modularized)</td>
</tr>
<tr>
<td>Display updating</td>
<td>(modularized)</td>
<td>Contains move notification</td>
</tr>
<tr>
<td>Move notification</td>
<td>Contained in FigureElement, AbstractFigureElement</td>
<td>Contained in DisplayUpdat­ing</td>
</tr>
<tr>
<td>Point bounding</td>
<td>Contained in Point</td>
<td>(modularized)</td>
</tr>
<tr>
<td>Logging</td>
<td>Contained in various</td>
<td>(modularized)</td>
</tr>
<tr>
<td>Factory enforcement</td>
<td>Contained in Point, Line</td>
<td>(modularized)</td>
</tr>
</tbody>
</table>

Table 3.14: Modularization Difference Summary

### 3.7.1 Name-Based Dependencies

Point bounding and tracing demonstrated a common difference involving name-based dependencies. A general characterization of the difference is shown in Figures 3.1 and 3.7.1. In the Java implementation, several methods caller₁, ..., all call the method callee. In the corresponding AspectJ code, a pointcut p names each of the caller₁ methods, and advice attached to p invokes callee. The overall effect in both cases is that callee runs either before or after any of the caller₁ methods run. The key feature of this pattern is that it removes dependencies from the callers, creating corresponding dependencies in the pointcut p. Several variants are possible depending upon how the elements are grouped into classes and aspects, so the class/aspect structures are not shown.

![Figure 3.1: Dependency differences created by using explicit method invocations](image)

A variant of this pattern is observed in the combination of basic move notifica-
tion and display updating. In this scenario, the Java implementation uses inheritance to obscure the identity of the concrete target method from the callers; the callers name an abstract method. At run-time, an object with a concrete implementation of this method is provided to the callers. In our example, basic move notification establishes the abstract interface (MoveListener.moved) and display updating provides the concrete target (DisplayUpdater.moved). The general AspectJ dependency structure is identical to Figure 3.1; the Java dependency structure is shown in Figure 3.2. This variant is more similar to the AspectJ dependency structure in that the callers are completely isolated from the ultimate callee method, but a dependency from each of the callers is still present.

![Diagram of dependency structure](image)

Figure 3.2: Using an abstract method to obscure the true identity of the invocation target

### 3.7.2 Whole-class Dependencies

Point bounding and basic move notification both demonstrate a common pattern of dependency difference. In the Java implementation, a whole-implementation dependency exists (e.g., Point depends on the rule “move methods must validate coordinates”) causing a col-
lection of method invocations (e.g., the set of calls from move methods to checkX). In the
AspectJ implementation, a whole-interface dependency exists (e.g., the pointMove pointcut
deps on all of the methods that move Point) causing a collection of methods to be named
(e.g., the pointMove pointcut names all methods in the current implementation that move
Point). A generalized pattern for this scenario is shown in Figures 3.7.2 and 3.7.2.

(whole impl)

Class c

Rule: all methods such that X
  must do Y

method m1
  (satisfies X)

method m2
  (satisfies X)

...  method y
  (realizes action Y)

Figure 3.3: Whole-class dependency addressed with explicit method invocations

3.7.3 Whole-Subsystem Dependencies

In the AspectJ implementation of basic move notification, a whole subsystem dependency
arises because a pointcut depends on all concrete implementations of the FigureElement
interface. In the Java implementation, the FigureElement interface includes a requirement
that all concrete implementations support basic move notification; this avoids a correspond­
ing whole-subsystem dependency. Therefore, this is a case in which a dependency in one
implementation (AspectJ) is absent in the other implementation (Java).

The factory pattern requires that all FigureElements be constructed within the
context of a factory method, and enforcement of this rule seems likely to introduce a whole-
subsystem dependency. However, neither the Java nor AspectJ implementation introduces
a whole-subsystem dependency: in both cases, the implementation is able to leverage the
existing inheritance structure instead.

3.7.4 Call-Graph Dependencies

This system has two examples of call-graph dependencies. In both cases, a single call-graph
dependency in the AspectJ implementation corresponds to several name-based dependen-
cies in the Java implementation. In both nested move suppression and factory pattern
enforcement, the AspectJ implementation has a single call-graph dependency arising from
the use of cflow or within, while the Java implementation has several whole-implementation
dependencies arising from the scattering of the concern. No call-graph dependencies were
present in the Java implementation.
Chapter 4

Implications of Dependencies

Dependencies have been a focal point of software engineers because they have a demonstrated impact on a variety of system properties[7][10]. In this section, we consider some of these properties: comprehensibility, re-usability and evolvability. Empirical study of these properties is beyond the scope of the present thesis, so only analytical consideration is provided.

4.1 Comprehensibility

Comprehensibility is the ease with which the system's code can be understood by a developer. In general, dependencies reduce the comprehensibility of the dependent modules[8], but not all to the same degree.

First, we consider the effect of name-based dependencies on comprehensibility. As an example, consider the point-bounding rule. In the Java implementation, the core Point class validated coordinates, creating a name-based dependency on the signatures of the methods that check the coordinate values. This dependency makes the core Point class less comprehensible, because a developer interested in the operation of the core Point class needs to understand the nature of the invoked coordinate-checking methods.

In contrast, the AspectJ version of Point has no dependency on point bounds check-
ing; instead, the dependency is reversed so that the `PointBoundsChecking` aspect depends on the `Point` class. In this case, we see a trade-off: the AspectJ Point class is more comprehensible at the expense of point bounds checking functionality.

The impact of whole-class, whole-subsystem and call-graph dependencies on comprehensibility cannot be established without some empirical evaluation. This exploration must be left for future research. However, we note briefly that explicit dependencies should improve comprehensibility simply because implicit dependencies are easier to overlook.

4.2 Evolvability

Evolution is the change to a system over time. Evolution may be done to enhance the performance of the existing system, to extend its functionality, or to adapt to changes in the environment (e.g., user interface trends or new platforms)[9]. In this section, we consider a subset of the evolution spectrum: we consider the range of changes that can be made independently, or changes to one module that do not require simultaneous changes to related modules. Coupled changes are problematic because they require more effort (making several modifications) and are potentially more error prone (if the affected dependents are not easy to find). This is the same range of evolutionary changes considered by Parnas [7][8]; this class of evolutionary change has the advantage of being analytically examinable.

First, we consider the impact of named-based dependencies on evolution. Named-based dependencies always constrain independent evolution in the same way: the name or signature of the referent cannot be changed without simultaneously changing the dependents. For instance, a method named `close` could not be changed to `release` without simultaneously changing the existing call sites that name `close`.

Next we consider the impact of whole-class dependencies on evolution. For instance, recall the `PointBoundsChecking` aspect in the AspectJ implementation. It depends on the `Point` class' whole interface, namely, on all methods that move `Points`. If a new move method is added to `Point`'s interface, for example, then the `PointBoundsChecking` aspect needs to be updated simultaneously.
The corresponding dependency in the Java implementation was the whole-implementation dependency: "all code that moves a Point must first validate the new coordinates." If a new move method is added to Point, it must be immediately updated to validate new coordinates. This is a kind of simultaneous change, and corresponds to the simultaneous change required by the AspectJ dependency described earlier.

Next we consider the impact of whole-subsystem dependencies on evolution. These dependencies tend to require simultaneous changes to the dependent when new modules with relevant characteristics are added to the subsystem. For instance, the DisplayUpdating.move pointcut in the AspectJ implementation is dependent on the collection of classes that implement the FigureElement interface. If a new FigureElement, Rectangle, is added, then the DisplayUpdating.move pointcut must be simultaneously modified accordingly.

Next we consider the impact of call-graph dependencies on evolution. These dependencies require simultaneous changes whenever the call graph is changed such that the existing assumptions are rendered invalid. For example, the nested move suppression rule assumes that any moves made in nested calls correspond to moves of nested objects. However, suppose that new functionality is added to the system so objects may be aligned relative to one another. In this scenario, the movement of one object may cause the movement of a non-nested peer object, and so the call graph dependency would be broken. This change would therefore require a simultaneous change to the DisplayUpdating.move pointcut.

4.3 Re-Usability

Re-use is an important goal of software engineering. It has the potential to decrease development time and expense while simultaneously increasing software quality[6]. The challenge for re-use, from a dependency perspective, is that all dependencies that a given module has must be satisfied in the target system.

For name-based dependencies, this means that the modules, methods, fields, etc. referenced by a module must be available exactly as named in the target system. Typically, name-based dependencies force any dependent modules to be included in the target system.
For instance, one could not use *PointBoundsChecking* aspect without the Point class, since it names *Point* explicitly.

Re-usable components are typically organized into libraries, and entire library is included in the target system. In Java and AspectJ, libraries are packages at the source level and JAR files at the binary level. Because of this pattern of re-use, any whole-class, whole-subsystem or call-graph dependencies that refer to elements included in the same library will always be satisfied without any cooperation by the target system.

If there are any dependencies that refer outside of the library, then these must be satisfied by the target system. For instance, if the library has a call-graph dependency that involves the call into the library, then the target system must be careful to avoid breaking this dependency. For example, the Swing user interface library included with JDK 1.2+ requires that any modifications to UI elements be made in the context of the Swing worker thread. Similarly, the Java security API is founded on a whole-system dependency that all classes which can do I/O in the entire system check with the security manager before allowing the I/O operation to proceed. Adding a new I/O class would violate Java's security model unless it obeys this dependency.
Key early work in this area was done by Stevens, Myers and Constantine[10]. The authors use the term “connection” in the sense that we use “dependency”. The authors state that, “a connection is a reference to some label or address defined (or also defined) elsewhere.” The connections that they considered are therefore what this thesis considers name-based dependencies.

This work also introduced the notion “coupling,” which is a connection plus an associated measure of “weight”. The authors proposed a weighting scheme, based on the nature of the interaction that gives rise to the coupling. For instance, if the caller can affect the callee’s control flow then the coupling is higher than if the caller is simply passing a data value to the callee. Subsequent work in software metrics has explored coupling in the context of object-oriented programming[2][1]. For instance, a method invocation between a subclass and superclass may have a different weight than a method invocation between two classes unrelated via inheritance. In this more recent work, the underlying dependencies remain name-based dependencies, from our perspective, but these dependencies are further categorized for the purpose of assigning coupling weightings.

This thesis has focused on the underlying dependencies, but it does not exclude the possibility of augmenting these dependencies with weights to produce measures of coupling. Furthermore, our grouping of all name-based dependencies into a single category does not
imply that all name-based dependencies are equivalent. Since it is not a goal of this thesis to assign weights to dependencies, the further subdivision of name-based dependencies is not necessary. Determining a weighting scheme for whole-class, whole-subsystem and call-graph dependencies, both within each type and relative to other types, remains a task for future work.

Parnas provided an early notion of the connections between modules[7] that is broader than the notion of coupling proposed Stevens et. al. Parnas proposed the following notion of connection: "the connections between modules are the assumptions which the modules make about each other." Parnas' notion of connection would capture all of the dependencies described in this thesis except for whole-subsystem dependencies (which are not relations between specific modules), although whole-subsystem dependencies would be captured by the spirit of his definition.

Similarly, the definition of a dependency in the Object Management Group's Unified Modeling Language[11] adopts a broad definition of dependency: "[a dependency] indicates a situation in which a change to the target element may require a change to the source element in the dependency." This definition captures all of the dependencies in this thesis except for whole-subsystem dependencies, which are captured by the spirit of the concept.

Therefore, both Parnas' "connection" and UML's "dependency" are essentially supersets of the dependencies discussed in this thesis. This thesis does not attempt to identify all of the kinds of dependencies that can exist in a software system; rather, the focus is on dependencies that seem to be important in the emerging field of Aspect-Oriented Software Development. So it should not be considered a failure that this thesis does not expand the scope of dependencies; rather, this thesis should be viewed as providing detail within the context of these concepts. Additional kinds of dependencies certainly do exist and may be identified more clearly in future work.
Chapter 6

Conclusion

6.1 Summary

This thesis proposes three additional dependencies to supplement the traditional notion of dependency in object-oriented systems. With whole-class dependencies, the dependent is sensitive to changes to certain changes to the class, but not restricted to particular parts of the existing class. For example, a pointcut may need to know all of the methods that move a Point; this pointcut has a whole-class dependency on Point because adding a new move method to Point would impact the pointcut. With whole-subsystem dependencies, the dependent is sensitive to certain changes to a package or packages, but not restricted to particular classes in the existing class. For example, a pointcut may need to know all of the kinds of FigureElements in the system; this pointcut has a whole-subsystem dependency because adding a new kind of FigureElement would impact the pointcut. With call-graph dependencies, the dependent is sensitive to changes to the system’s call graph. For example, an aspect may assume that all nested invocations of FigureElement move methods correspond to movements of logically nested FigureElement objects; this pointcut has a call-graph dependency because moving a non-nested FigureElement in a move method would impact the pointcut.
6.2 Future Work

These dependencies are intended to be useful to developers in evaluating system designs, in the same way that traditional name-based dependencies are used by developers currently. Some additional work is needed to explore the application of these dependencies for system design evaluation and comparison. We also hope to be able to use these dependencies as a basis for comparing implementations in different programming languages.
Bibliography


