Project Plan/Proof-of-Concept: Refactoring an Open-source System with AspectJ

Tony Kong
University of British Columbia
Vancouver, BC
y2k0b@ugrad.cs.ubc.ca

Raymond Situ
University of British Columbia
Vancouver, BC
r7p0b@ugrad.cs.ubc.ca

Kevin Wong
University of British Columbia
Vancouver, BC
b2j0b@ugrad.cs.ubc.ca

Benjamin Hwang
University of British Columbia
Vancouver, BC
r6o0b@ugrad.cs.ubc.ca

James Yoo
University of British Columbia
Vancouver, BC
l4k0b@ugrad.cs.ubc.ca

ABSTRACT

AspectJ is a powerful extension to the Java language which enables developers to apply a separation of concerns to a system with a high degree of crosscutting and scattering. This report details our refactoring of a small open-source project [1] (a Java implementation of Space Invaders) using AspectJ, and describes a possible approach to refactoring a larger-scale software system with the lessons learned and insights provided from our initial work.

CCS CONCEPTS

• **Software and its engineering** → *Frameworks*;

KEYWORDS

Software evolution, refactoring, aspect-oriented programming

ACM Reference Format:
Tony Kong, Raymond Situ, Kevin Wong, Benjamin Hwang, and James Yoo. 2018. Project Plan/Proof-of-Concept: Refactoring an Open-source System with AspectJ. In *Proceedings of Intro. PL (CPSC 311).* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.475/123_4

1 OVERVIEW

After much consideration, our group has decided to achieve the 90% completion level for this project. This means that this report will place greater detail and emphasis on the proof-of-concept (POC) we have produced and provide a high-level description of the project for our 100% completion target. We detail the specific aspects we have created to mitigate the crosscutting concerns in our POC, and the specific issues that they helped resolve in the system.

2 PROOF-OF-CONCEPT OVERVIEW

In our background report [1], we stated that we would utilize AspectJ to handle the crosscutting concerns of exception handling and logging in a software system. These concerns were present in the prototype system in the form of logging statements whenever a resource was loaded/a thread was created, or when an exception was thrown because of an IO error, or when a resource failed to load. Instances of these situations were scattered throughout the system, and provided us with numerous points where we could apply aspects. As a result, our refactored POC system contains the following aspects.

[1]https://github.com/jyoo980/aop-spaceinvaders

CPSC 311, 2018W1, UBC Vancouver
2018. ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

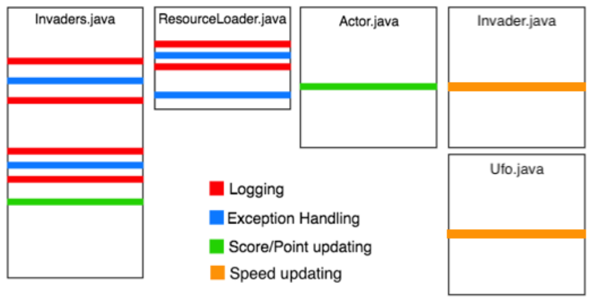

Figure 1: crosscutting concerns found in aop-spaceinvaders

(1) `ExceptionsHandler.aj`
   - handles `InterruptedException` or any exceptions related to resource loading.
(2) `InvadersLogger.aj` and `ResourceLogger.aj`
   - eliminates scattered log statements from the system, a classic use of Aspect-oriented programming.
(3) `Scoreboard.aj`
   - a domain-specific use of aspects, all scattered score-keeping logic is encapsulated into this aspect.
(4) `UpdateSpeed.aj`
   - another domain-specific aspect use, handles the update logic for the x/y speed of game objects.

3 ARRIVING AT THE ASPECT

This section will detail the development and refactoring process we undertook to complete the POC.

3.1 Identifying Crosscutting Concerns

This was possibly the easiest part of the development of the POC. We knew exactly what we were looking for in terms of classical concerns which could be targeted by aspects; namely, logging and exception handling. We also found domain-specific use cases of aspects in the form of score update logic, and x/y position update logic. A high level representation of the crosscutting nature of these concerns is displayed in *figure 1*. The process of identifying these concerns was aided by the Eclipse IDE and its AspectJ Development Tools, which was an immensely powerful facility for visualizing crosscutting and where advice was applied in our refactored POC.

3.2 Identifying Join Points and Pointcuts

Identifying join points in our crosscutting concerns was the most creative component of the POC. This was due to the sheer number of patterns which we could use to match points on our program. We then composed these join points into pointcuts which we then applied advice to. In this section, we document our approach to extracting join points which we then utilized to form our pointcuts for our aspects.

3.2.1 **Logging**

We have a method addInvaders() in a class Invaders

```
public void addInvaders() {
    // Boilerplate omitted
    for (int i = 0; i < rows; i++) {
        for (int j = 0; i < cols; j++)
            Invader inv = new Invader(this);
    }
  LOGGER.info("Created " + rows * cols + " invaders.");
}
```

We wanted to remove the logging line in this method. We could easily identify the join points for the pointcut we would use to capture this log statement, which were:

- within(Invaders)
- execution(* Invaders.addInvaders(..))

It is important to note that the above two join points are only a small subset of other equally valid ones which could have been used instead. We then compose the join points above into a valid pointcut, which is described below:

```
pointcut addingInvaders():
    within(Invaders) &&
    execution(* Invaders.addInvaders(..));
```

After the composition of the pointcut addingInvaders(), we would then start to create our advice. This will be detailed later, along with our process in creating aspects, in section **3.3.1**

3.2.2 **Exception Handling**

There were a number of exceptions thrown in the aop-spaceinvaders system. We detail one here as an example of how we picked out join points for our exception-handling aspects for our POC. We have the following method in our class ResourceLoader. The code is shown below:

```
public void getSprite(String name) {
    // Boilerplate omitted
    try {
        // Boilerplate omitted
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

This is a particularly interesting case since there are many ways to pick out join points for this exception handling. We decided to select join points such that they would be very specific as to when our advice would execute. Our join points are shown below:

- execution(* ResourceLoader.getSprite(..))
- args(name)
- target(rLoader)

The first join point is fairly standard, highlighting a time in the control flow of the system when getSprite(..) is executed. The second, args(name) is interesting since it allows us to pick out the exact argument which getSprite(..) is called with. The primitive join point target(rLoader) is also powerful since it enables us access to the calling object in the body of our advice. We then composed these three join points into the pointcut below

```
pointcut spriteExn(String name, ResourceLoader rLoader):
        execution(* ResourceLoader.getSprite(..)) &&
        args(name) &&
        target(rLoader)
```

This pointcut is one of the most complex we've designed, and its corresponding advice will provide interesting insight into AspectJ's powerful mechanisms for aspects which are able to perform elegant error handling. We explore its corresponding advice in **3.3.2**

3.2.3 **Score Tracking**

The domain-specific logic related to score tracking in Space Invaders pushed us to identify join points and compose pointcuts outside of the usual realm of logging and exception handling. We had the method below which was called whenever a collision occurred between the object upon it was called and another, in class Actor

```
public void setMarkedForRemoval(boolean remove) {
    this.markedForRemoval = remove;
}
```

This is an extremely important method. The class Actor is a superclass which many, if not all of the objects in aop-spaceinvaders extend. Because of this, there were a large number of calls to this method effectively scattered throughout the system, and not just when each time the score for the game is updated. This is exactly the type of situation that Kiczales et al described when they said that they "have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement." [2] - i.e. it is a problem which OOP cannot cleanly modularize. We captured calls to this method with the following join points

- call(void Actor.setMarkedForRemoval(..))
- args(isMarked)

We then composed the following pointcut with the join points we selected above:

```
pointcut hit(boolean isMarked):
    call(void Actor.setMarkedForRemoval(..)) &&
    args(isMarked)
```

Although this appears to be a simple pointcut, its utility will become evident in **3.3.3** when we create advice which uses it.

3.3 Advice and Aspects

In the previous section, we explored how we picked out join points and composed pointcuts for our crosscutting concerns. We now detail the advice we apply in our refactoring process to integrate AspectJ into `aop-spaceinvaders`. Specifically, we explore the development process of `InvadersLogger`, `ExceptionsHandler`, and `Scoreboard`.

3.3.1 Logging

In **3.2.1**, we created the following pointcut from the join points we picked out to represent the logic which added invaders into the game during startup:

```
pointcut addingInvaders():
    within(Invaders) &&
    execution(* Invaders.addInvaders(..));
```

We want to log the total number of invaders that were instantiated in the game, exactly like how it is done without using aspects in the previous version. In order to do this, we create another pointcut:

```
pointcut invaderConstructor():
    call(Invader.new(..));
```

With these two pointuts in place, we can now construct the aspect `InvadersLogger.aj`:

```
privileged aspect InvadersLogger {
    private int invaders = 0;
    private final Logger Log = Logger.getInstance();

    // Pointcuts are as shown in 3.2.1

    after(): invaderConstructor() {
        this.invaders++;
    }

    after(): addInvaders() {
      Log.info("Created: " + this.invaders + " invaders");
    }
}
```

There are a few things of note here: firstly, the full source code of this aspect and all other aspects mentioned in this section are available in the online repository `aop-spaceinvaders`[2], and the second is the fact that there are a number of similarities between a Java class and an aspect. For example, just like a class, an aspect is permitted to have class members with the usual access modifiers (private, public, protected, etc...). The reserved keyword `privileged`, enables the `InvadersLogger`aspect to have access to the private members of classes referenced by our pointcuts. This is important because it allows for greater flexibility in handling crosscutting concerns while still maintaining access control and requiring a minimal number of changes, if any, to the original classes. There is another aspect which handles logging, `ResourceLogger`, but for the sake of brevity, an exploration into that will be omitted in lieu of the source code being available online to view. We now move on to a discussion of the aspect for exception handling.

[2]https://github.com/jyoo980/aop-spaceinvaders

3.3.2 Exception handling

This section continues the discussion of utilizing aspects to handle exceptions which was started in **3.2.2**. We'll use the same pointcut, `spriteExn(..)` to inform our discussion here. In our POC, we still want to maintain the same level of robustness as we had before, but with the removal of `try-catch` mechanisms from any non-aspect code. We arrive at the aspect `ExceptionsHandler`:

```
privileged aspect ExceptionsHandler {

    // Pointcuts are as shown in 3.2.2

    declare soft: IOException:
        execution(* ResourceLoader.getSprite(..));

    BufferedImage around(String n, ResourceLoader rl):
        spriteExn(n, rl) {
        try {
            proceed(n, rl);
        } catch (IOException e) {
            // Log Here
        }
        return rl.images.get(name();
    }
}
```

Again, some parts of `ExceptionsHandler` are omitted for brevity and the full source code may be viewed online. We have the statement:

```
declare soft: IOException:
    execution(* ResourceLoader.getSprite(..));
```

This is a construct of the AspectJ language which enables us to remove `try-catch` mechanisms around a line of code which throws an exception, in this case, we will not have to surround a call to `getSprite(..)` since we have declared it to be "softened". Note that this does not mean an exception is *not* thrown. It is indeed thrown, but it is now allowed to be intercepted by our `around` advice. Note that in the body of our advice, we execute the body of the original method by calling `proceed(n,rl)`.However, we no longer require a `try-catch` block within the original body of `getSprite(..)`.

3.3.3 Score Tracking

Last, but not least, we return to our domain-specific use of aspects for our Space Invaders game. We had a remarkably simple pointcut which captured points in the control flow when a collision occurred between an object and another:

```
pointcut hit(boolean isMarked):
    call(void Actor.setMarkedForRemoval(..)) &&
    args(isMarked)
```

However, we leverage this simple pointcut to create the aspect below:

```
public aspect Scoreboard {

    private int score = 0;
    // Pointcuts same as 3.2.3
    after(Actor actor, boolean isMarked):
        target(actor) && hit(isMarked) {
        if (isMarked) {
            this.score++;
        }
    }
}
```

This `after` advice is an *amazingly* powerful piece of code. It will execute after each point which matches with the pointcut `hit`. This will effectively match any and all collisions between two objects which extend the `Actor` class. This is an example of how aspect-oriented programming used in conjunction with object-oriented programming enables developers to write some very powerful programs.

3.4 Results

By composing together a number of available join points, pointcuts, and advice, we have arrived at a relatively well-formed exploration of some basic aspects. We have created aspects for error handling, logging, and an aspect for our domain-specific concern of keeping track of and updating our game score. In **4**, we explore what effect, if any, these aspects had on the overall quality of our system.

4 METRICS

In this section, we use the software metric *Dedication (DEDI)* [3] to measure the effect that the use of aspects had in our system. DEDI is given by the formula below:

$$DEDI(t, s) = \frac{\text{SLOC in component t related to concern s}}{\text{SLOC in component t}}$$

Where SLOC refers to *Source Lines Of Code*. This equation will form the basis for our basic static anaylsis in reasoning about the effects of aspects in our system. For the purposes of this POC, each component will be a method that is now advised by an aspect, and the concerns explored will be the ones mentioned in previous sections, that is: logging, exception handling, and score updating.

4.0.1 Logging

We will look at a few methods (components) that are now advised by either of the aspects `InvadersLogger` or `ResourceLogger`.

(1) `InputHandler.handleInput(KeyEvent event)`
- Advised by: `InvadersLogger.aj`
- SLOC related to logging prior to refactoring: 2
- Total SLOC prior to refactoring: 10
- DEDI prior to refactoring = 20%
- SLOC related to logging after aspects: 0
- DEDI after refactoring = 0%

This is an example of where refactoring using AspectJ was particularly successful; the component no longer has any lines of code which are related to a crosscutting concern (in this case, logging). Therefore, the dedication of this method with respect to logging is now 0.

(2) `ResourceLoader.getSprite(String name)`
- Advised by: `ResourceLogger.aj` and `ExceptionsHandler.aj`
- SLOC related to logging prior to refactoring: 2
- Total SLOC prior to refactoring: 14
- DEDI prior to refactoring: 14.28%
- SLOC related to logging after aspects: 0
- DEDI after refactoring = 0%

In this method, we observe a decrease in a method's dedication to logging from 14.28% to 0%. We are able to remove all instances of logging from our `getSprite` method by having two different aspects handle the different crosscutting concerns related to our logging logic.

4.0.2 Exception Handling

Metrics obtained from using aspects to encapsulate exception handling are described below. This was arguably the most powerful use of aspects and the aspect-oriented programming paradigm in our POC, as it *completely* eliminated the use for `try-catch` mechanisms in our code.

(1) `Invaders.game()`
- Advised by: `ExceptionsLogger.aj`
- SLOC related to exceptions prior to refactoring: 3
- Total SLOC prior to refactoring: 32
- DEDI prior to refactoring: 9.4%
- SLOC related to logging after aspects: 0
- DEDI after refactoring = 0%

To fully illustrate what was enabled by the `ExceptionsLogger` aspect, we detail a small portion of the code before refactoring:

```
try {
    Thread.sleep(timeDiff/100);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

The following is the identical call to `Thread.sleep(..)` after the introduction of the `ExceptionsLogger` aspect:

```
Thread.sleep(timeDiff / 100);
```

All of the exception handling logic is now delegated to the aspect, removing the crosscutting `try-catch` mechanism from the `game()` method. Other concrete examples can be seen in the source code hosted online.

4.0.3 Score Tracking

We detail the result of using aspects to target domain-specific crosscutting concerns in this section. It is clear that even for non-canonical applications of aspect-oriented programming with AspectJ, we can still improve code quality:

(1) `Invaders.updateWorld`
- Advised by: `ScoreBoard.aj`
- SLOC related to scoreboard updates prior to refactoring: 3
- Total SLOC prior to refactoring: 17
- DEDI prior to refactoring: 17.6%
- SLOC related to scoreboard upates after refactoring: 0
- DEDI after refactoring = 0%

Notice that through the use of aspects, we were able to remove the crosscutting calls to the update logic present in our `updateWorld()` method and others.

5 EXTENDING BEYOND THE PROOF-OF-CONCEPT

In the previous sections, we discussed the application of aspect-oriented programming to a small open-source software system via AspectJ. This was a proof-of-concept which served to showcase the mitigating effects which aspects have toward a system's cross-cutting concerns. This section will detail additional features we would implement in our system. The low-risk approach details how we would go about implementing some, but not all of these features. The high-risk approach is comprised of the completion of the low-risk approach and some additional work.

5.0.1 The Low-risk Approach

A low risk approach to extending our project beyond the POC phase would be to select a larger open source software system and apply the same approach to it as we did with our POC. That is, we would:

(1) Identify crosscutting concerns.
(2) Generate aspects to capture said concerns.
(3) Obtain software quality metrics from utilizing aspects.

We would utilize the same *DEDI* metric as we used in our POC, with an additional metric proposed by Murphy et al:

$$DOF(t) = \frac{|S| \sum_{S}^{S}(DEDI(t,s) - \frac{1}{|S|})^2}{|S| - 1}$$

DOF, or *Degree of Focus* is a measure of "how well concerns are separated in a component" [3]. This would enable us to better reason about the effects of refactoring a system with AspectJ.

5.0.2 The High-risk Approach

This approach would take our project to a far larger scale. We would select a project from one of the open-source projects as we described in our background report: Mockito, Guava, or IntelliJ. Then apply the same aspect-oriented refactoring process with an important additional component:

(1) Run the jPeek [3] Java static anaylsis tool on our large-scale project.

The jPeek tool is an automated application which enables a developer to statically derive a number of metrics. We would focus on the available metrics below:

- Class cohesion
- Method cohesion

We would use these metrics to inform our study of AspectJ and the effects it has on an open source system at scale. We believe that this would be an appropriate reach goal because of two reasons:

(1) The need to get familiar with and refactor an *industrial* grade software system, e.g. Mokito, Guava, IntelliJ.
(2) The need to deploy and run an unfamiliar tool (jPeek) on a refactored system.

The reasons above constitute a significant amount of time and work invested into this project, and would constitute the full 100% goal of our deep refactoring of an open source system using AspectJ.

[3] https://github.com/yegor256/jpeek

REFERENCES

[1] AspectJ, A Pragmatic Approach to Cross-cutting Concerns *CPSC 311, 2018W1.*
[2] Kiczales et al, 1997: *Aspect Oriented Programming*. European conference on object-oriented programming, 220-242
[3] Murphy et al, 2007: *Identifying, Assigning, and Quantifying Crosscutting Concerns* Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques