Background Report: AspectJ, a Pragmatic Approach to Cross-cutting Concerns

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

This paper details AspectJ, a general-purpose Aspect-oriented programming language built on top of Java. We explore AspectJ's mechanisms that enable software developers to encapsulate cross-cutting concerns. We also detail how the language allows developers to better reason about their systems by enforcing a separation of concerns between business logic and cross-cutting logic. Finally, as a practical example of the power of AspectJ and Aspect-oriented programming, we describe our project, where we refactor a small open-source version [1] of the classic game *Space Invaders* using AspectJ.

CCS CONCEPTS

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

KEYWORDS

AspectJ, Aspect-oriented Programming, Pointcut, Joinpoint, Advice, Cross-cutting concern, application logic, business logic

ACM Reference Format:
Tony Kong, Raymond Situ, Kevin Wong, Benjamin Hwang, and James Yoo. 2018. Background Report: AspectJ, a Pragmatic Approach to Cross-cutting Concerns. In *Proceedings of Intro. PL (CPSC 311)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.475/123_4

1 OVERVIEW

The average size of a software system is growing each year. As technology takes a more central role in the lives of many people around the world, software begins to permeate across many aspects of life. Analogous to this growth is the inevitability that logic is being included in a system to fulfill application logic as opposed to business logic. Application logic may range from simple behaviour like logging, to more complex components such as security implementations and exception handling. At its core, application logic relates more to the infrastructure of a system as opposed to what the system should aim to fulfill, or its business requirements.

When unrelated logic becomes scattered throughout a program, we see an evolution of that logic into a cross-cutting concern. We say that it is cross-cutting because it cuts across multiple components of a program without belonging to any single one of them. This tangling [6] of cross-cutting concerns in a system becomes problematic for many reasons, two of which we will highlight below due to their relevance to our project

(1) Program cohesion decreases
(2) Encapsulation of cross-cutting concerns becomes more important

AspectJ addresses the two issues above by providing a mechanism to encapsulate cross-cutting concerns, and an even more powerful mechanism that enables developers to specify what behaviour programs should have when handling these concerns.

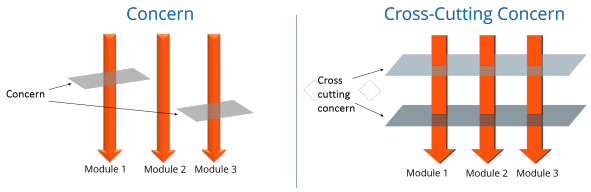

Figure 1: concerns in a software system

2 THE LANGUAGE

2.1 Introduction

AspectJ can be thought of as an extension to the Java programming language. This does not mean, however, that it is devoid of its separate syntax and signature. In fact, AspectJ enables the clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols [1]. These features are enabled by the *Aspect*, and the various components which it is comprised of.

2.2 The Aspect

At the core of AspectJ, or any Aspect-oriented programming language is the *Aspect*. The Aspect is the basic unit of modularity and Aspect-oriented Programming [7], and can be considered to be the embodiment and the entity which encapsulates a system's cross-cutting concerns. We declare the aspect like so:

[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

```
aspect <id> { ... }
```

where `<id>` is some valid identifier name in the AspectJ language. Note the similarity to the syntax of Java class declarations:

```
class <id> { ... }
```

Although similar in syntax to how one would declare a class in Java, this is where Java classes and AspectJ aspects begin to diverge. An aspect may contain entities of type `Pointcut` and `Advice`, these types are unique to AspectJ and will be explored later in this section. We explore the aspect `InstantiationAspect`below

```
aspect InstantiationAspect {
    int count = 0;

    pointcut new_exec(): execution(new(..));

    before(): new_exec() {
        count++;
        System.out.println("new was called");
    }
}
```

We first highlight the main differences between a class and an aspect.

(1) An aspect cannot be directly instantiated [2]
(2) Aspect implementation can cut across other types [2]

(1) is extremely important because it is, at the core, descriptive of how aspects work. Because they cannot be directly instantiated, aspects fulfill their core logic by being *weaved* throughout the program in which they are declared. In fact, one can and should think of the statement

```
aspect <id> { ... }
```

As declaring *and* creating an instance of an aspect, akin to the statement

```
class <id> { ... }
<id> instance = new <id>(...);
```

in Java. Note that an aspect exists as a singleton instance in a program. Once you declare an aspect, the logic you declare inside of it will execute as your program does without any additional work. Below are some of the similarities that aspects share with classes and the concepts it borrows from OOP

(1) Aspects may extend classes and implement interfaces (note that this does *not* enable aspects to be instantiated. [2]
(2) Aspects may extend other Aspects

Looking back at `InstantiationAspect`, we have the statement

```
pointcut new_exec(): execution(new(..));
```

This is an example of the type `Pointcut`. Generally, a pointcut is an aspect's way of encapsulating a certain execution point. In this case, the pointcut `new_exec()` represents the point(s) in a program where a constructor is executed. This facility for encapsulating points in a program's execution is remarkably powerful, but its true usefulness is enabled by advice.

Advice in AspectJ is compactly described in the informal EBNF below

```
<Advice> ::= <Before>
           | <After>
           | <Around>
```

Join Point	Current Object	Target Object	Arguments
Method Call	executing object*	target object**	method arguments
Method Execution	executing object*	executing object*	method arguments
Constructor Call	executing object*	None	constructor arguments
Constructor Execution	executing object	executing object	constructor arguments
Static initializer execution	None	None	None
Object pre-initialization	None	None	constructor arguments
Object initialization	executing object	executing object	constructor arguments
Field reference	executing object*	target object**	None
Field assignment	executing object*	target object**	assigned value
Handler execution	executing object*	executing object*	caught exception
Advice execution	executing aspect	executing aspect	advice arguments

* There is no executing object in static contexts such as static method bodies or static initializers.

** There is no target object for join points associated with static methods or fields.

Figure 2: available AspectJ join points

Advice can be combined with pointcuts to dictate the runtime behaviour of a system at a certain point in time. This is exactly what happens in `InstantiationAspect`

```
before(): new_call() {
    count++;
    System.out.println("new was called");
}
```

This advice will execute the system print statement before each point in the program which matches the pointcut `new_call()`. Other types of advice will be discussed later. With this brief overview of the Aspect, we are now able to understand `InstantiationAspect`. If we wanted to enable the same behaviour in a program without AspectJ; it would be difficult - one possible implementation may require a global counter and a increment of this counter after *all* calls to `new`. This is certainly not an example of evolvable software design, and may not even be feasible for large systems.

2.3 Joinpoints

The JoinPoint is a crucial entity of AspectJ and Aspect-Oriented Programming in the sense that it enables a sytematic categorization of some points of a program's execution. Formally, a join point is a "well-defined point in the execution of a program." [4]. The keyword here is *well-defined* - without formally defining which points are in a program are valid/invalid join points, it would be extremely difficult for a developer to understand and reason about an Aspect-oriented system [6]. Figure 2 details the join points which are enabled by AspectJ.

Notice that we already saw an example of constructor execution in our `InstantiationAspect` with our pointcut `new_exec()`. The wide variety of join points enable a developer to target a variety of points in their programs where they want to execute some cross-cutting code. A classic example would be the update logic for the observer design pattern [10]. If we had a GUI interface with some capability for drawing shapes, it is very well likely we could have some method such as

```
public setXPoint(int x) {
    this.x = x;
    this.update();
}
```

The call to `this.update()` is a cross-cutting concern. With AspectJ, a developer could target `setXPoint` as an instance of a *method execution* or *field assignment* join point. This enables the developer to formally define points in a program where they could then encapsulate in pointcuts, which will then be used to specify when advice in a system will be executed. This will be explored later.

2.4 Pointcuts

While the join point enables the developer to reason about points in program execution where advice could be possibly applied, pointcuts are points in the program where advice *will* be applied. One can think of pointcuts as being a subset of join points in a program. To construct a pointcut, the developer also provides a predicate expression that Join points in the execution flow can match against. These predicate expressions contain signature patterns that are intended to describe the type of Join points a pointcut matches against. Signature patterns can describe things such as a methods name, the return type of a method, the type of arguments it takes, the number of arguments it takes, and more. AspectJ has support for more than 10 pointcuts, we will discuss only a few here.

2.4.1 Method Call and Execution

This represents Pointcuts of the form

```
call(MethodPattern)
```

The EBNF describing `MethodPattern` [5] is much too large to include here, so we show a number of examples instead. This Pointcut represents when a method is called in a program. For example, the following pointcut declaration will match calls to any methods in class `Main` which return something of type `int`

```
pointcut any_main_call(): call(int Main.*(..));
```

We can of course have more complex `MethodPatterns`.

```
pointcut example():
    call(void com.ajexamples.*.foo(*));
```

This pointcut will match a call to a method called foo, which must return void, and is found in any class which is located in the package com.ajexamples. We also have a powerful way to compose any number of pointcuts via logical operators. Like below

```
pointcut composite(): any_main_call() && example();
```

The `composite` pointcut will match pointcuts which match the two previously declared above. The example for `execute` will be largely identical, save for the way in which it is declared

```
execute(MethodPattern)
```

and the circumstances during which the match will occur, method *execution* vs method *call*. There is a subtle but important difference between the `execute(..)` and the `call(..)` pointcut type. Although developers new to AspectJ may deem these pointcuts as the same, it is important to note that `execute(..)` will match when the *body* of the `MethodPattern` described in execute is actually being run, whereas `call(..)` represents a point in time when the specific `MethodPattern` has been *called*.

2.4.2 Exception Handling (within and handler)

The set of pointcuts below are *extremely* powerful facilities that AspectJ provides with respect to exception handling

```
within(TypePattern)
handler(TypePattern)
```

`within` enables a match with any of the code which resides inside the type specified by `TypePattern`, while `handler` forces a match for any code which throws an exception matching the `TypePattern`.

```
pointcut null_handler():
    within(Main) && handler(NullPointerException);
```

With the `null_handler` pointcut, we effectively perform a match on the code in the class `Main` which handles exceptions of type `NullPointerException`. It is important to note that `within` is not specific only to exception handling. It is a remarkably versatile construct which allows a finer, more focused level of granularity when picking out pointcuts from a large number of joinpoints. For example, we could have a pointcut which matches a large number of joinpoints in class `A` and `B`. In this specific instance, a developer can match a subset of the joinpoints by specifying either `within(A)` or `within(B)` to match only the joinpoints residing in those specific classes.

2.5 Advice

With some basic background in the Aspect, Joinpoint, and Pointcut, we now arrive at *Advice*. Without advice, there would be nothing determining how our code runs at every join point selected by a pointcut. Therefore, we use advice to communicate exactly how the code should execute and run to address the cross-cutting concern. We return to the informal EBNF describing the types of advice enabled by AspectJ

```
<Advice> ::= <Before>
           | <Around>
           | <After>
```

From this EBNF, we see that there are three types of advice: `before`, `around`, and `after`. At a high level, `before` will execute code *before*, `after` will execute code *after*, and `around` will execute code *around* a selected pointcut. These three types of advice are explored in the following sections.

2.5.1 Before

At a basic level, we declare `before` like so

```
before(): <Pointcut> { ... };
```

There are multiple cases when this particular advice would be useful. A classical example of a prudent use of before would be to lock a mutex before control flow enters a critical section of code. It would be similar to the code below

```
before(): concurrent_access() {
  System.out.println("Entering Critical Section");
    mutex.lock();
}
```

Applying this advice to the `concurrent_access` pointcut would enable developers to enforce a separation of concerns between the logic they are attempting to execute with the concurrent code, and the boilerplate logic they must execute at each invocation of said concurrent code. We have already seen an example of `before` in our `InstantiationAspect`.

2.5.2 After

Similarly to `before`, we declare `after` like so

```
after(): <Pointcut> { ... };
```

In our previous example, we had advice which would lock a mutex before we entered a critical section in our concurent code. Utilizing the power of `after`, we can delegate the unlocking of the mutex to a single point in our code.

```
after(): concurrent_access() {
  System.out.println("Exiting Critical Section");
   mutex.unlock();
}
```

Notice that with the advice from the previous section, the developer is now capable of handling all mutex-based concurrency control mechanisms simply in the bodies of these aspects. This is a huge step in separating the concern of concurrency control, which would usually cut across multiple modules, from core business logic.

2.5.3 Around

With `before` and `after` in place, we can now discuss the third, and arguably, most powerful type of advice - `around`. We can declare an `around` advice like so

```
<TypePattern> around(): <Pointcut> { ... };
```

Notice the `<TypePattern>` present in the declaration of the advice. This means that unlike `before` and `after`, `around` is able to *return* actual values. This will be explored later, but we first return to our concurrency example from the previous section. Without `around`, we would have to make use of both `before` and `after` to encapsulate the cross-cutting concurrency mechanisms, like so

```
aspect MutexAspect {
   pointcut concurrent_access():
       execution(<some concurrent code>);
   before(): concurrent_access() {
       mutex.lock();
   }
   after(): concurent_access() {
       mutex.unlock();
   }
}
```

This is an example of a cross-cutting concern which is an ideal candidate for refactoring using `around`, since they represent calls which will happen in pairs which close the `concurrent_access` join point between themselves. We refactor `MutexAspect` to make use of `around` like so

```
void around(): concurrent_access() {
    mutex.lock();
    proceed();
    mutex.unlock();
}
```

`proceed()` is an optional call which will execute the control flow of the original pointcut. This brings us to a critical aspect of `around`, which is the fact that it is able to execute logic *in lieu* of the original pointcut. We use an example from Baeldung [8]

```
class Atm {
   public bool withdraw(int amount) {
       this.funds -= amount;
       return true
   }
}
aspect AtmAspect {
   pointcut withdraw_cash():
       call(bool Atm.withdraw(*));

   bool around(int amount): withdraw_cash() {
       if (amount <= Atm.MAX) {
           return proceed(amount);
       }
       return false;
   }
}
```

With this example, we illustrate the power of `around`. Not only is it able to take override the control flow of the original program, but it also has the ability to *return* the flow of control to point in the program which is matched by the pointcut.

3 LANGUAGE LIMITATIONS

AspectJ, even with its powerful mechanisms of enabling a developer to encapsulate and separate cross-cutting concerns, is an imperfect language - like all others [2]. In this section, we highlight what we believe to be the most significant pitfall of AspectJ.

3.1 Comprehension Indirection

When designing large systems, developers will have to reason about the nature of their programs in order to find ways to write code that integrates well into the system as a whole. The most popular way for developers to understand a system by far is to perform a *static analysis* - this often consists of a developer reading through methods or functions and reasoning about what they do. Take the simple method for example below

```
public bool withdraw(int amount) {
   if (amount <= this.MAX_AMOUNT && amount > 0) {
       this.cashReserve -= amount;
       return true;
   }
   return false;
}
```

A developer can statically reason about the dynamic behaviour of this code quite well; it will return false when some error condition is fulfilled, otherwise it will subtract `amount` from the cash reserve and return true. If we now rewrite the above in AspectJ with the following advice:

```
bool around(int amount): call_withdraw() {
   if (amount > MAX_AMOUNT || amount <= 0) {
```

[2] except for Racket, which is a perfect language.

```
        return false;
    }
    return proceed(amount);
}

public bool withdraw(int amount) {
    this.cashReserve -= amount;
    return true;
}
```

We arrive at a rather serious problem. The developer is no longer able to reason about the dynamic behaviour of withdraw just by performing a static read-through of the method itself. It is now *much* easier for the developer to misunderstand what this method is truly doing without looking at both the method itself, and its related aspect.

This means that reasoning about the nature of a program statically falls prey to a high level of indirection - a developer now has to consider both a system *and* its aspects as separate but interweaved entities when attempting to understand it. This becomes even more problematic when AOP is applied to even larger systems with a large number of distinct aspects.

4 PROJECT OVERVIEW

Having discussed the AspectJ language and its features in detail, we may now explore how we are going to utilize it in our final project. Below is a high-level overview of the issue we will attempt to solve with AspectJ and aspect-oriented programming.

4.1 The Problem

Consider the system that is described in *figure 3*. We have a concern, λ which is scattered throughout a number of classes. We describe λ as *cross-cutting*, since it cuts across multiple modules of the system. This is a common situation in modern software systems. In fact, application logic such as logging, and exception handling [6] appear extremely often in most software systems, and are prime examples of cross-cutting concerns.

4.2 A Proposed Solution

To tackle the problem of *crosscutting*, we will apply Aspect-oriented programming principles to a system via AspectJ. Eventually transforming the system described in *figure 3* to a system represented by *figure 4*. We will accomplish this transformation by refactoring a system using AspectJ. Specifically, we will perform the following refactorings

(1) Identify and quantify cross-cutting concerns in a software system.

These will be measured using metrics such as *concentration* [9] and others developed by Murphy et al.

(2) Create aspects to capture the concerns identified in (1)
(3) Declare pointcuts to capture join points where cross-cutting logic appears.
(4) Declare advice to execute the handling of each pointcut.

Parts (2) to (4) will be the critical focus of the project, requiring us to demonstrate a strong command of the newly introduced AspectJ language constructs and the aspect-oriented programming paradigm.

4.3 Targeted Concerns

We will target application logic related to the following concerns in our selected system

(1) Logging
(2) Exception handling
(3) Assertions and condition validation

(1) and (2) are canonical examples of crosscutting concerns, and are exceptional candidates for modularization via aspects. (3) is particularly interesting because it covers a very wide range of possible conditions. It covers situations ranging from simple assertions on some values during program execution, to a completely different branch being taken depending on some previous validation. We expect this will make excellent use of the `around` advice.

4.4 Candidate Projects

We have three candidate open source projects.

(1) Mockito: https://github.com/mockito/mockito
(2) Guava: https://github.com/google/guava
(3) Intellij: https://github.com/JetBrains/intellij-community

All three of our candidate projects have a large number of our target cross-cutting concerns, and are widely used. An analysis and refactor of any of our candidates will provide insight into the number of cross-cutting concerns in a developed system and highlight the positives and negatives of AspectJ.

4.5 The 100% Project in Detail

If we were to accomplish 100% of our project, we would chose to refactor Mockito using AspectJ. The reason for this is because of the fact that it is an industrial-grade open-source project with rich documentation, and because it contains the classical candidates for AspectJ's use cases. Examples include the large number of user-defined exceptions in the project, as well as the `MockitoSessionLogger` class which could be easily refactored into an Aspect.

4.6 The Target Project in Detail

We will refactor a smaller open source project [3] which is an implementation of Space Invaders in Java. The reasons why we chose this project as opposed to the 100% project are as follows

(1) Familiarity with the domain
(2) Smaller, and manageable project size
(3) Ease of setup for development

Although we undertake a refactoring of a smaller project, it is important to note that we will still be leveraging AspectJ in the same way we would have done for the 100% project. That is, we will mainly be targeting exception handling, logging, and in this case, display update logic with Aspects.

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

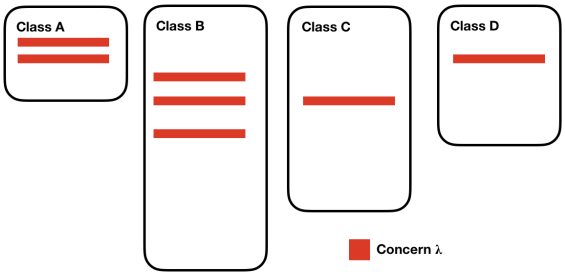

Figure 3: scattering of a concern in a software system

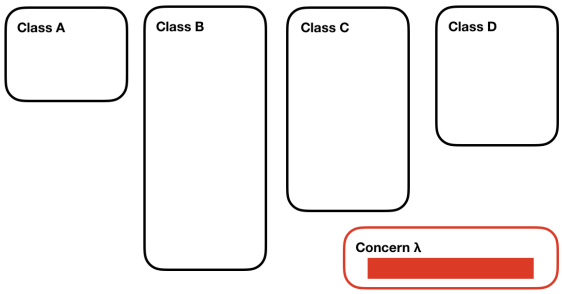

Figure 4: system with an encapsulated cross-cutting concern

4.7 Anticipated Results

After we perform our refactor of the system using AspectJ, we expect that less concerns in the system will be crosscutting. We will investigate this hypothesis by re-analyzing the system with the same methods as previously described.

In our results, we also plan to highlight the limitations of AspectJ as previously mentioned in section **3**. We believe that this will be a point of interest in attempting to understand and reason about the design and developer compromises associated with using AspectJ, and aspect-oriented programming in general.

REFERENCES

[1] The Eclipse Foundation *Introduction to AspectJ.* `https://www.eclipse.org/aspectj/`
[2] AspectJ - The Aspect, The Eclipse Foundation *Appendix B. Language Semantics.* `https://www.eclipse.org/aspectj/doc/next/progguide/semantics-aspects.html`
[3] AspectJ - Advice, The Eclipse Foundation *Appendix B. Language Semantics.* `https://www.eclipse.org/aspectj/doc/next/progguide/semantics-advice.html`
[4] AspectJ - Join Points, The Eclipse Foundation *Appendix B. Language Semantics.* `https://www.eclipse.org/aspectj/doc/next/progguide/joinPoints.html`
[5] AspectJ - Pointcuts, The Eclipse Foundation *Appendix B. Language Semantics.* `https://www.eclipse.org/aspectj/doc/next/progguide/pointcuts.html`
[6] Kiczales et al, 1997: *Aspect Oriented Programming.* European conference on object-oriented programming, 220-242
[7] Kiselev, Ivan, 2002: *Aspect-Oriented Programming with AspectJ.*
[8] Baeldung, accessed: 2018/11/06 *https://www.baeldung.com/aspectj*
[9] Murphy et al, 2007: *Identifying, Assigning, and Quantifying Crosscutting Concerns* Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques
[10] Kiczales, Gregor: *Aspect Oriented Programming: Radical Research in Modularity* Google Tech Talks, YouTube: https://www.youtube.com/watch?v=cq7wpLI0hco