11/24/2014

Final Project Thesis

Instance Pointcuts

George de Heer
S1366696

1st Supervisor:
Christoph Bockisch

2nd Supervisor:
Mehmet Aksit

Table of Contents

Abstract

In this thesis I will be looking at two challenges; the main goal being to implement a language extension named Instance Pointcuts, based on an implementation strategy defined by Kardelen Hatun, and give case studies to show the benefits of the language compared to their original Java implementation. To support this research, first I will be implementing a modular and extensible AspectJ language that offers a clear and simple structure from which new languages can be derived.

The purpose of the Instance Pointcuts language extension is to be able to select objects within an application during its execution based on its participation in certain events; not simply it's class type. This functionality is useful in several scenarios, mostly when a specific class' objects are accessed regularly during runtime, where these pointcuts allow us to pick out specific objects that have followed a certain pattern, and interact with them separately from the other objects of the class.

I aim to demonstrate and further prove the usefulness of the Instance Pointcuts language through the use of case studies that will be implemented using our language specification, which will also demonstrate how our AspectJ implementation can be easily used to create new languages of some level of complexity.

Implementing the case studies in the new Instance Pointcuts language does show that there are benefits of the language over plain Java, which I have demonstrated in both a code comparison, as well as the resultant ecore models. Instance Pointcuts in these scenarios have shown to not only simplify the structure of the application, especially the classes to which I focussed on rewriting in the new language, but also it reduces the lines of code Instance Pointcuts required, and the amount of code scattering throughout classes is lowered. This results in the total effort required to develop and maintain the application to be significantly reduced.

1. Part I

1.1 Introduction

In software development, when creating an application in a popular Object-Oriented language such as Java, we often require classes to interact with each other and reference one another. While this requirement is supported by the language; in a big project this will in most cases lead to scattered and tangled code across the system. This leads to increased complexity and difficulty to understand the system, and also reduces the ability to perform maintenance.

Aspect-Oriented languages such as AspectJ do aim to solve this problem by using aspects to monitor and adapt a program when certain events of importance happen, allowing the developer to move their otherwise tangled code to the more modular aspect which sits on top of the project. One feature Aspect-Oriented languages lack by default however is the ability to look at the history of an object, and be able to store an object or collection of objects based on their usage pattern during execution.

This feature, which I will be referring to as Instance Pointcuts, can be implemented currently in Java code or in an application using an existing Aspect-Oriented language. This however comes at a cost of likely requiring a lot of invasive code, depending on the size and complexity of the application, to be able to find such objects based on their usage pattern, since such event information may be scattered throughout the application.

The Instance Pointcuts feature would remove the need for writing invasive code into the application by adding a new type of pointcut to aspects which themselves aim to be modular and simply monitor or adapt an application. This new type of pointcut would be able to select and store objects based on their participation in certain events, and allow the application to interact with these collections of objects – giving object interaction on a finer level of granularity.

The code necessary to implement these pointcuts within an application would usually be placed within a class-like entity called an Aspect, as defined by AspectJ. This greatly reduces the scattering of the code that would otherwise occur when implementing the same functionality in traditional Java methods, even though these pointcuts are optionally allowed to be written in Java classes as well. Furthermore, there is evidence, provided by Kardelen Hatun's study [24], that amount of effort necessary to implement these pointcuts is significantly lower than in Java; partly because the language benefits from the functionality provided by AspectJ. Instance Pointcuts however provide reusability and compose-ability, as well as managing the objects in a multiset behind the scenes. This make the language both easy to use as well being powerful in regards to amount of code necessary to achieve the desired functionality.

Implementing this functionality as well as finding use cases that prove its effectiveness and usefulness is the main goal and focus of this thesis. This functionality will be created within a language called Instance Pointcuts, which already exists in a prototype stage defined by Kardelen Hatun [24]. I will then be using the Instance Pointcuts language to implement some existing projects, written originally in, in my defined language instead, and comparing the two implementations to find out if, and to what extent, the application benefits from being written in the Instance Pointcuts language.

Since this language is built as an extension of AspectJ, my first challenge will be to establish a modular AspectJ language which is as feature-complete as possible and supports extensibility. While currently there are already several implementations of an AspectJ compilers, including a prototype Kardelen Hatun has defined for the purpose of the Instance Pointcuts language, I must first assess if any of the current compilers meet the requirements of modularity and extensibility that we seek, and if not, what approach we should take as a foundation for defining our own AspectJ language.

Once the AspectJ compiler has been chosen, implemented, and tested, I can then move on to my second challenge which would be to define the Instance Pointcuts language based on Kardelen Hatun's prototype and strategy for implementation. While I would aim to make the language as feature-complete as possible as defined by the strategy, the most important requirement is to make sure what I do implement is correct, and furthermore that it can be transformed back into our AspectJ implementation.

1.2 Background on Object SelectIon

In the study I presented in [20], an in-depth literature study was made on the topic of Instance Pointcuts. In this study I presented both the current progress in the implementation of the language as done by Kardelen Hatun, as well as an analysis into other areas of related work where the goal was being able to refer to and interact with objects based on more than just their type.

These related works are relevant to my research since they present different approaches that may improve the modularity of code within an application, even if they do not cover all the functionality that we are interested in. While we may also argue that within some applications and under some circumstances our Instance Pointcuts language may be superior to these approaches; for my focus I have only looked at these approaches as alternatives to my implementation of instance pointcuts.

In my study I did an investigation into the topics of Eos [21], OQL [22], and Typestates [23], which all offer different methods of accessing objects as well as offering a varying amount of functionality in interacting with the objects. This study did show that only the Instance Pointcuts language offers the functionality of selecting objects based on their history; so we are bringing something new to the table. There is however some future potential of the Instance Pointcuts language, in regards to being combined with one of these languages, such as a combination with Typestates to offer verification of the correctness of objects that we select with our pointcuts.

A large focus of my previous study was to find potential applications for the language; which is an important part of this project as well. In Part III I will be looking into detail on case studies for the language; which will be java applications were found based on specific heuristics I defined as criteria which potential applications had to meet.

1.3 Terminology

- **Instance Pointcuts** – A specific type of pointcut that we define in the Instance Pointcuts language, which allows us to select an object or set of objects based on their participation in an event. An event can be defined for the instance pointcut for when an object may be added to a multiset collection which the instance pointcut represents, and also an event may be defined for when an object is removed from the collection. An instance pointcut is also defined with a type unlike regular pointcuts which do not declare one.
- **Instance Pointcut Expressions** – There are two types of defined instance pointcut expressions; an add expression and a remove expression. An instance pointcut consists of one add expression and an optional remove expression. An instance pointcut expression contains either an after-

event or before-event, or both, to find objects that will be added or removed from the instance pointcut.

- **Events** – There are two types of events, which similar to advices in AspectJ are named before, and after, representing at what point in time regarding the event the object is exposed to the instance pointcut expression. An event's body consists of a pointcut expression as defined in AspectJ which will locate objects of relevance to the instance pointcut.
- **Multisets** – These are a specific type of collection or set, which behave like ordinary sets, but allow for elements within the set to be of a multiset type themselves, leading to the ability to have sets within sets. This is a necessary feature instance pointcuts require, as often multiple objects will need to be added to the collection at once, and also to support composite instance pointcuts these multisets would have to be able to be combined while still retaining their original structure.
- **Composite Instance Pointcuts** – These are another new type of pointcut which as the name suggests are made up of a composition of two or more instance pointcuts. The composite instance pointcuts do not have any instance pointcut expressions themselves; instead they monitor the set operations of the instance pointcuts it is composed of, and updates its own set accordingly. Furthermore the type of the composite instance poincut has to be type compatible with all the types of the instance pointcuts it is composed of.
- **Refined Expressions** – With the introduction of composite instance pointcuts, this new feature of refining instance pointcuts gains significance. As instance pointcuts can be referenced by other instance pointcuts; the referenced pointcut will be composed of a refinement expression. There are several levels of refinement that can be done to the referenced instance pointcut:
    - An instance pointcut can be referenced directly by just its name, and the refinement expression will take all the before/after events of the add/remove expression of the referenced instance pointcut. For our implementation of the Instance Pointcuts language, this is the only level of refinement we will support.
    - An instance pointcut could however also be referenced by its name and add/remove expression, or even also its before/after event, which provide different levels of access to the instance pointcut, and offer a finer granularity of refinement of the expression.

1.4 Research Questions

In this section I will first be presenting the main research questions I will be answering in this thesis; since the study presented in [20] they have not been modified, only one new question (4) was added since some research was required to find a suitable AspectJ compiler. Apart from that, the focus of the project has remained the same so the original questions still stand. Question 1 is the main research question to be answered, with questions 2, 3 and 4 being the sub research questions that are required to answer the main question.

1. Can Instance Pointcuts provide improvements or advantages to the quality of the code of a program over the same program written only in an object-oriented or other aspect-oriented language?
2. How can any improvements an application written in the new Instance Pointcuts language has over the original application written in plain Java or AspectJ be measured?
3. How will I find suitable candidate programs that can be rewritten in the new language and compared with the original program?

4. Is there an already existing AspectJ compiler, which is both modular and extensible, that can be used as a base for the Instance Pointcut language to extend upon, or will it be necessary to write our own?

1.5 Solution Approach

To answer the Research Questions I defined, the first milestone of the project consisted of establishing an AspectJ implementation that could be used and extended upon for the Instance Pointcuts language extension. As it was decided I collaborate with Hristofor Mirchev on this part since he required a similar AspectJ implementation for his project; we as a team agreed on implementing our own extensible AspectJ language definition using Model Driven Engineering techniques.

Along with defining the language, we also decided to do testing on the implementation to make sure AspectJ models could be derived from it and the syntax and semantics of the language was correct. While testing does not have much significance to my individual research, we still decided to do it to make sure we implement a sound and reusable AspectJ language.

On top of that, we also had to write the transformation from AspectJ into Java so that applications can be compiled. While there are several methods of doing a transformation from one language to another; as we were using MDE techniques for developing the language, it made sense to use a Model-to-Model transformation language.

Once the AspectJ implementation has been finalised the focus could be shifted back to the Instance Pointcuts language extension. Similar to the AspectJ implementation, first the language needs developing using MDE techniques, as well as writing the transformation back to AspectJ.

After that, the case studies can be done, by first selecting Java applications that have potential to benefit from being implemented in the language extension. Once they have been selected they could be implemented in the Instance Pointcuts language. Instance pointcuts may only apply to certain areas of the application, so overall, depending on the application, a part of it will still be kept in its original form.

The first step to implement the case study in the new language is to write all the new syntax for the class or classes that I am aiming to improve. This can be achieved through running the language's resource project in a new Eclipse instance [13]. From that instance, the existing case study project can be imported and the new syntax can be defined and overwrite the existing Java code where necessary. Since the syntax rules are defined in the concrete syntax of the language; it is very strict how the Instance Pointcut entities are written. While I do aim to reduce the code scattering in the application, I do not want to introduce any invasive Java code alterations unless it is necessary; so it is good to remove any redundant code, but not introduce any further code scattering.

Once the new syntax has been written for the application, I can then convert the textual syntax to an Ecore model which is used for the transformation stage into AspectJ and Java. The model can then be transformed and compiled and run just like the original, and a comparison can be made on how beneficial it was for the application to be written in Instance Pointcuts. This comparison will occur as both a code comparison between Instance Pointcuts textual syntax compared to the Java syntax, but also a model comparison between the ecore model before transformation, and after transformation into AspectJ. As the goal of the thesis is not to compare the AspectJ to Java transformation, there is no need to make a comparison between the AspectJ and Java models; comparing the Instance Pointcut model is a more meaningful comparison to judge the benefits of my defined language. A conclusion can then be drawn based on an analysis determining what the benefits are of the Instance Pointcuts language over the original Java.

1.6 Thesis Structure

The remainder of the thesis can be broken down into two parts; the first part about implementing the AspectJ language and the second part about the Instance Pointcuts extension.

1.6.1 The AspectJ Implementation

This part of the thesis discusses the collaborated work between Hristofor Mirchev and I, where we first will give an introduction and background information on the language and current implementations of AspectJ, as well as our approach for implementing the language.

After that we will discuss our design choices for the implementation of the language and how we came to our final structure for the language. Since the language is built on top of Java, we have also had to define a transformation into Java code so it can be compiled, and finally we will also be discussing the testing we have done to get a degree of correctness for the language.

1.6.2 The Instance Pointcuts Extension

In this part I will be presenting my individual contributions in the form of not only implementing the Instance Pointcuts language extension on top of AspectJ, but also proving its worth through the use of case studies. These case studies will all be existing applications written in Java, and re-implemented in the new language. A comparison will then follow to prove the language fulfils its design goals, as well as to what degree it grants benefits to the application against a set criteria.

Finally a conclusion will be drawn from this analysis, and a mention of the remaining future work will be given, both in terms of what is remaining to achieve a feature complete Instance Pointcuts language and the benefits this would have, but also further extensions of the language.

2. Part II

AspectJ Metamodel Implementation and Transformation to JaMoPP

Collaboration of Hristofor Mirchev & George de Heer

2.1 Background

This chapter's contributions were divided roughly equally between myself (George de Heer) and Hristofor Mirchev, with the sections on AspectJ and part of EMFText written by myself, and the sections on Model Driven Engineering and half of EMFText written by Hristofor Mirchev.

2.1.1 AspectJ

Aspect-oriented programming (AOP) is a paradigm that strives to increase modularity in programming by separating pieces of the program that rely on or affect other parts of it. While object-oriented programming (OOP) offers us a way to modularize common concerns, AOP offers a way to modularize cross-cutting concerns in particular.

AspectJ is a Java extension that supports AOP. It adds a few new concepts to Java. A *join point* is a point in the program flow. Examples of join points include method calls, method executions, constructor executions, object instantiations and others. A *pointcut* is a predicate on a set of join points that selects a subset of them. *Advice* is code meant to be implicitly executed when the program flow encounters a join point matched by a pointcut. Finally, an *aspect* is the module that encapsulates all these new constructs. Another feature of AspectJ called *inter-type declarations,* which affects the static structure of the program, namely it allows the programmer to add fields, methods or interfaces to existing classes, is outside the scope of our research.

2.1.2 Model-driven engineering

Model-driven engineering (MDE) is a system development methodology that focuses on the creation and use of *models*. Models are abstract representations of the concepts related to a specific problem domain. A model can describe the entities in a domain, their attributes, relationships between them and constraints on those relationships. Models are specified using some notation, described in a *modeling language.* This notation usually consists of at least a description of the *abstract syntax* (i.e. the concepts and relationships) and a description of the *concrete syntax* (i.e. the physical appearance of those concepts and relationships). The abstract syntax is commonly defined through a *metamodel*. All the terminology and considerations for models are applicable for the metamodel as well, as the metamodel is just a model of the model.

MDE strives to increase productivity in at least three respects. First, by raising the level of abstraction MDE closes the semantic gap for domain experts that may otherwise not be experienced in programming with a general-purpose language. Second, MDE tools can impose constraints and perform *model validity* checks to detect and prevent errors early in the development cycle. Model validity is the process of evaluating the model against different criteria, either coming from the metamodel or in the form of constraints, written by the programmer. Lastly, through the use of *code generation* and *model transformation*, MDE increases "automation" in software development thereby limiting the possibility of human errors. Code generation is the process of generating source code from the model while model transformation is the act of transforming a source model into a target model through the use of transformation rules. This is one of the primary reasons why we find MDE to be particularly good at implementing a language conforming to our design goals. Most MDE tools generate a lot of the complex components (i.e. lexers and parsers) for the programmer, which leaves him with the task of implementing only a few highly modular elements of the metamodel. Extensibility, on the other hand, can easily be achieved through the reuse of the metamodel or model transformations.

Frameworks for building applications and whole systems using models are called *modeling frameworks* or *language workbenches*. A very mature and popular modeling framework is the Eclipse Modeling Framework (EMF). Its metamodeling language (Ecore) is based on EMOF (a standardized metamodeling language) and it provides easy to use tools for code generation for EMF models that lay the grounds for interoperability with other EMF-based applications.

2.1.3 EMFText / JaMoPP

EMFText is a language workbench for defining textual languages, be it domain-specific (DSL) or general-purpose (e.g. Java) based on Ecore metamodels. It provides a rich DSL for syntax specification – the *Concrete Specification Language* (CS) based on EBNF, that can generate an editor with features like syntax highlighting and code folding, and components to parse and print instances of the metamodel. The general development process with EMFText consists of the following steps:

1. **Specifying the abstract syntax for the language (the .ecore metamodel)**
   To define a language's metamodel we must consider how to break down the language into what *entities* it consists of, what *attributes* do they have and what are the relations or *references* between them. References have to further be distinguished into *containment* and *non-containment* references. A containment reference relates an element of the model (a *parent*) with another, defined in the same context (a *child*). A non-containment reference relates a model element with one that is defined somewhere else. Let us consider the example of modeling a standard Java class. The class entity would hold a containment reference for a method declaration inside the class, but a non-containment reference to a method for a method call statement since the method can be defined elsewhere.
2. **Specifying the concrete syntax for the language (the .cs file)**
   Having completed the metamodel we continue by defining the textual representation of all its entities in the .cs file. The .cs file can be roughly broken down to two sections. The first one contains metadata for the language. This can be information on what the file extension for the language is going to be, what is the root element of the language, what are the tokens (to help the lexer tokenise the input correctly) and how to highlight them, and some code-generation instructions. The second part of the .cs file contains the syntax rules for the language. A rule is a textual representation of a specific entity in the metamodel with its attributes and references. Various other concepts such as keywords, operators for multiplicity (*, ?. +) and brackets for nested sub-rules are also regularly used in a syntax rule.
3. **Generating the tools for the language**
   After defining the syntax specification, we can use the EMFText generator to create the accompanying language infrastructure. This includes the Java-based implementation of the metamodel, a parser and printer, reference resolvers that resolve names of non-containment references and classes related to an Eclipse-based editing functionality like syntax highlighting.
4. **(Optional) Customizing the tools for the language**
   The previous step generates a basic tooling for the language. However, EMFText offers ways in customizing it with additional advanced features like code completion, code folding, refactoring,

semantic validation post parsing and more. For languages where the trivial reference resolvers, generated in the previous step, are not sufficient, EMFText also provides means for writing custom ones that override the behaviour. This can usually be the case in general-purpose programming languages where references can span cross-resources like Java or AspectJ.

The *Java Model Parser and Printer* (JaMoPP) is a complete implementation of Java 5 in EMFText. It offers a metamodel covering the whole language, a text syntax conforming to the Java specification and custom-written reference resolvers that correctly capture the Java static semantics when cross-referencing metaclass entities.

2.2 AspectJ Implementation

Before getting into the discussions of our AspectJ implementation in detail, let us examine the existing open AspectJ compilers and how they align with our design goals. This section's main contributor was myself (George de Heer), with the AspectJ-Front section written by Hristofor Mirchev.

2.2.1 AspectBench Compiler

The AspectBench compiler (abc) is a complete implementation of theAspectJ 5 language that "aims to make it easy to implement both extensions and optimizations of the core language" [8]. It is based on the Polyglot [9] and Soot [10] frameworks.

A simplified overview of the workflow of abc begins when the Polyglot parser parses the input .java source file into an abstract syntax tree (AST). It then runs a series of transformations to separate the AST in two parts. One part holds only the pure Java constructs, while the other contains the additional AspectJ information like the advice bodies, inter-type declarations and others. The process is then taken over by the Soot framework. It takes the purely Java AST and transforms it to its internal representation called Jimple. The framework then uses several modules that can convert freely between Jimples, Java byte code and Java source code to conduct the weaving process and output the final .class and .java files.

Compared to our approach, using abc has two disadvantages. First, Polyglot offers a clean and modular way to extend the grammar, but has a standard lexer for interpreting it, which is not extensible. This means that to make abc recognize the new language we would need to extend the existing grammar and copy, and rewrite the existing lexer. Since the lexical analysis in AspectJ is already complicated enough due to pattern matching, for example, rewriting the lexer for the extension will be an intimidating task.

This is not the case in our approach, as the CS language in EMFText allows us to reuse the lexer written for JaMoPP. Secondly, we argue that the weaving process in our approach is handled with less effort. The declarative way in which we "carry" the AspectJ information to be woven by using the annotation-based style is easier to understand than Polyglot's transformations that separate its AST. This also enables us to delegate the complexity of the weaving process to AspectJ's load time weaver, something that cannot be done in abc, rather than having to implement it in Soot ourselves.

2.2.2 AspectJ-Front

AspectJ-front is the combination of a syntax definition and a printer for AspectJ 5 and is made using the Spoofax modeling framework. The syntax definition is written in SDF, which is the metamodeling language in Spoofax. The printer is built in Stratego/XT. This is a subset of tools in Spoofax used specifically for program transformations. The printer is written as transformation rules that change the initial parse result (stored in Spoofax's ATerm format) into text.

AspectJ-front by itself is an extension of a similar combination of syntax definition and printer for Java called Java-front. This already shows that extensions can be written with relative ease which matches one of our design criteria we set out to achieve - extensibility. AspectJ-front is also modular as the syntax definition is clearly separated from the printing rules. Thus, it also matches the second criteria we have set. The reason why we did not opt to use it is because it did not match the third design goal we have –

We argue that picking EMF with respect to simplicity is better than the alternative of using Spoofax for two reasons. First, building an extension on top of previous work in EMF would require extending both the Ecore metamodel and the CS syntax definition. Having done that, EMF generates a parser and printer for you. The same result in Spoofax would require extending the SDF syntax definition, but also writing a new printer. One might argue that for the additional cost of writing the printer by hand, Spoofax at least skips the metamodeling step, however this is not truly the case. A programmer still has to have a mental image of the metamodel to follow when writing the syntax definition. EMF simply externalizes that process and produces a tangible artifact (i.e. the .ecore file) that can be shared and used as a specification between programmers. Secondly and more importantly, we believe that starting out with Spoofax in general is the harder approach. The complexity comes from the framework itself. Spoofax is not mature and lacks proper documentation. The framework is outdated and barely supported anymore. To develop an extension in Spoofax you have to understand its internal representation of models (i.e. ATerm), the syntax definition language (i.e. SDF) and the Stratego/XT toolset it provides. This combined with the lack of proper documentation can be a daunting task. EMF, on the other hand, is still supported, has an extensive documentation and requires less EMF-specific knowledge to get started.

2.2.3 ReflexBorg

"The ReflexBorg approach is a method for implementing aspect-oriented extensions ofJava, including both their syntax and semantics" [11]. It consists of three layers. One layer is for the syntax definition of the language, written in SDF. Another is for the transformation of the abstract terms of the aspect language into Java code instantiating Reflex elements, which is written in Stratego. The final one takes care of the semantics and weaving and is written in Reflex. Reflex is a Java implementation of a versatile kernel for aspect-oriented programming using bytecode transformation.

ReflexBorg uses the same metamodeling language (SDF) and the same model transformation tool (Stratego) as AspectJ-front, so the same concerns regarding the maturity of these tools apply here as well. Another reason to stick to our approach is that, similarly to the abc case, ReflexBorg forces us to implement the weaving logic which we would otherwise avoid.ReflexBorg does have some limitations in that it does not support all features of AspectJ such as advice execution and exception handlers. It also does not provide a type checker, and currently expects input to be of type-correct state or incorrect code would be generated.

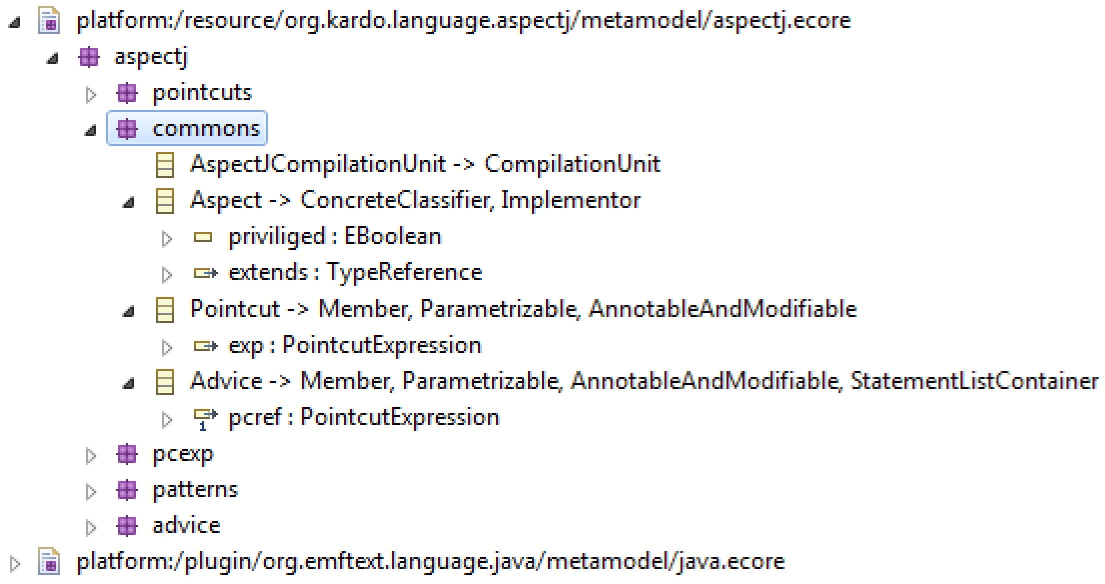
2.3 Defining our AspectJ Metamodel

This sections main contributor was myself (George de Heer), with Hristofor Mirchev writing the Patterns section. In implementing of the AspectJ language however this part was divided equally, with my focus being on Pointcuts, Pointcut Expressions and Advices, and Hristofor Mirchev's focus being on Patterns and the Commons package.

Following the naming convention set out from Kardelen, whose AspectJ prototype we used as inspiration, our metamodel consists of five subpacakges:

1. *Commons* package - contains entities for the top-level first class members in AspectJ (i.e. aspect, pointcut and advice).
2. *Pointcuts* package – contains entities for the 18 primitive pointcuts that the language supports.
3. *PcExp* package – contains entities for the acceptable pointcut combinators. Those are && (and), || (or) and ! (negated).
4. *Advice* package – contains entities for the 5 different advice types (i.e. Before, After, After Returning, After Throwing and Around).
5. *Patterns* package – contains entities for the pattern matching in pointcuts.

2.3.1 The Commons Package



```
platform:/resource/org.kardo.language.aspectj/metamodel/aspectj.ecore
  aspectj
    pointcuts
    commons
      AspectJCompilationUnit -> CompilationUnit
      Aspect -> ConcreteClassifier, Implementor
        priviliged : EBoolean
        extends : TypeReference
      Pointcut -> Member, Parametrizable, AnnotableAndModifiable
        exp : PointcutExpression
      Advice -> Member, Parametrizable, AnnotableAndModifiable, StatementListContainer
        pcref : PointcutExpression
    pcexp
    patterns
    advice
platform:/plugin/org.emftext.language.java/metamodel/java.ecore
```

Figure 1. The Commons Package Structure

The key design consideration here that drove the design of the whole package was whether an aspect should be treated "exactly" like a regular Java class. According to the AspectJ Developers Guide [1] an aspect declaration is syntactically similar to a class definition. Three of the differences they point out are that an aspect can cut across other types; that it cannot be directly instantiated and that in case of nesting, the nested aspect must be static. However, the difference that influenced our metamodel design was that a class cannot contain advice code.

Advice extending from the Java class ‘Member’ is of course not behaviour that should be allowed; since Advices should never be able to be written in a Java class just like a method. Pointcut definitions at least are allowed to be written in a Java class. As we don’t want the prohibited behaviour of Advice to occur, with this design choice it forces us to write a check to prevent it in a post-processing stage by doing a semantic check. We ended up opting for this design choice for two reasons:

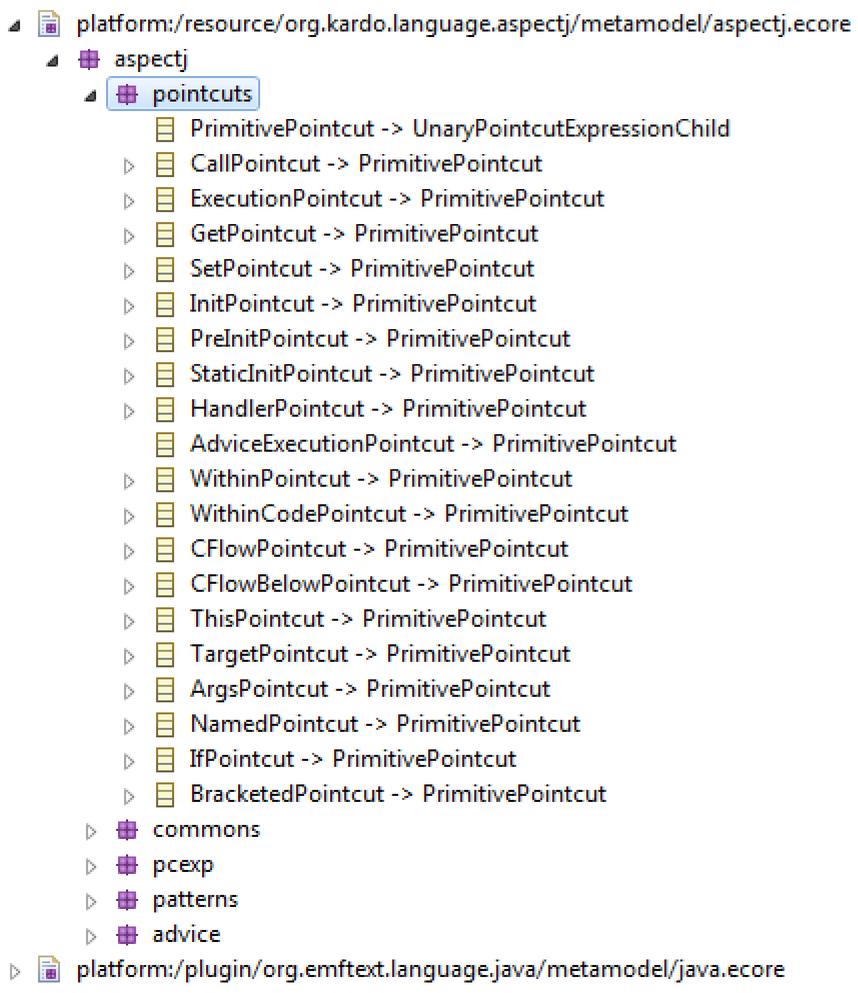
1. First, it simplifies the metamodel. Enforcing such a constraint in the metamodel would result in an overhead of entities. We would need to make one entity that “captures” all the class contents (e.g. methods, fields and pointucts) and another for the Advice. This differentiation would also result in a need for two more entities so that a compilation unit can hold both aspects and classes. Without this overhead the metamodel is more streamlined.

2. Secondly, the *JaMoPP* model that we are using as a foundation comes with setter and getter methods implemented for the different entities it has. Directly extending these entities lets us reuse these methods which eased the process of implementing the custom reference resolvers for our AspectJ implementation. Another important structural decision we made in the commons package was to make Pointcut and Advice exist within the package rather than be stored within their respective packages. While this does not affect the behaviour of the metamodel, it would have made the overall structure be more confusing. This is because especially for Pointcuts, the Pointcut class is among 20 other classes, and not seen clearly on a higher level.

The AspectJ compilation unit directly extends the Java compilation unit. Since it contains a reference to ConcreteClassifiers*,* we make the aspect extend *ConcreteClassifier*. We also directly extend *Implementor* to complete the functionality of an aspect to implement another. It contains one attribute of type boolean to determine if the aspect is priviliged or not and one reference to a type, in case the aspect extends another. The *ConcreteClassifier* entity contains a reference to *Member,* which is a supertype for the different class members (e.g. methods, fields). Thus, we model the pointcut and advice as Members. Both advice and pointcut entities contain a reference to a pointcut expression.

Finally, the *AspectJCompilationUnit* inherit from Java’s *CompilationUnit*, which is another design choice we made; the alternative would be to inherit directly from Java’s *JavaRoot*. As this would mean not inheriting the *CompilationUnit’s* fields or methods and having to specify them itself, our choice avoids this.

2.3.2 The Pointcuts Package



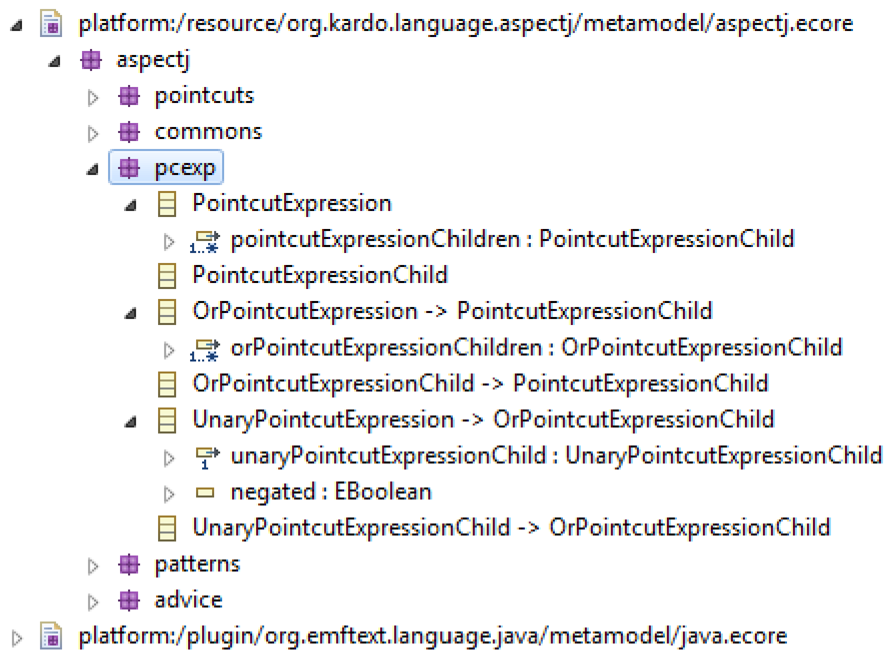
```
platform:/resource/org.kardo.language.aspectj/metamodel/aspectj.ecore
  aspectj
    pointcuts
      PrimitivePointcut -> UnaryPointcutExpressionChild
      CallPointcut -> PrimitivePointcut
      ExecutionPointcut -> PrimitivePointcut
      GetPointcut -> PrimitivePointcut
      SetPointcut -> PrimitivePointcut
      InitPointcut -> PrimitivePointcut
      PreInitPointcut -> PrimitivePointcut
      StaticInitPointcut -> PrimitivePointcut
      HandlerPointcut -> PrimitivePointcut
      AdviceExecutionPointcut -> PrimitivePointcut
      WithinPointcut -> PrimitivePointcut
      WithinCodePointcut -> PrimitivePointcut
      CFlowPointcut -> PrimitivePointcut
      CFlowBelowPointcut -> PrimitivePointcut
      ThisPointcut -> PrimitivePointcut
      TargetPointcut -> PrimitivePointcut
      ArgsPointcut -> PrimitivePointcut
      NamedPointcut -> PrimitivePointcut
      IfPointcut -> PrimitivePointcut
      BracketedPointcut -> PrimitivePointcut
    commons
    pcexp
    patterns
    advice
platform:/plugin/org.emftext.language.java/metamodel/java.ecore
```

Figure 2. The Pointcuts Package Structure

In the Pointcuts package there are many types of pointcuts which all inherit from PrimitivePointcut, which makes the structure of the package very clear and simple – even while having all of the 18 AspectJ pointcuts being supported within it [1]. Such a design allows for an easy implementation of pointcut extensions. For a new type of primitive pointcuts, a programmer can add a new entity that extends PrimitivePointcut or he can modify it to introduce new global pointcut functionality.

Another design choice we made was not only to move the Pointcut class to the Commons package as described earlier, but to not make it an abstract class that could be used to support different types of pointcuts. This would mean that in AspectJ an instance of a Pointcut would have to be made from a child-class of Pointcut that would have to be defined. This does allow extensions of the language to also define new types of pointcuts without inheriting properties that are specific to AspectJ ones. However, even though we want to support modularity and extensibility of the language, we do not want our implementation to be tailored to specific languages, so we decided to not have Pointcut as an abstract class.

2.3.3 The PcExp Package



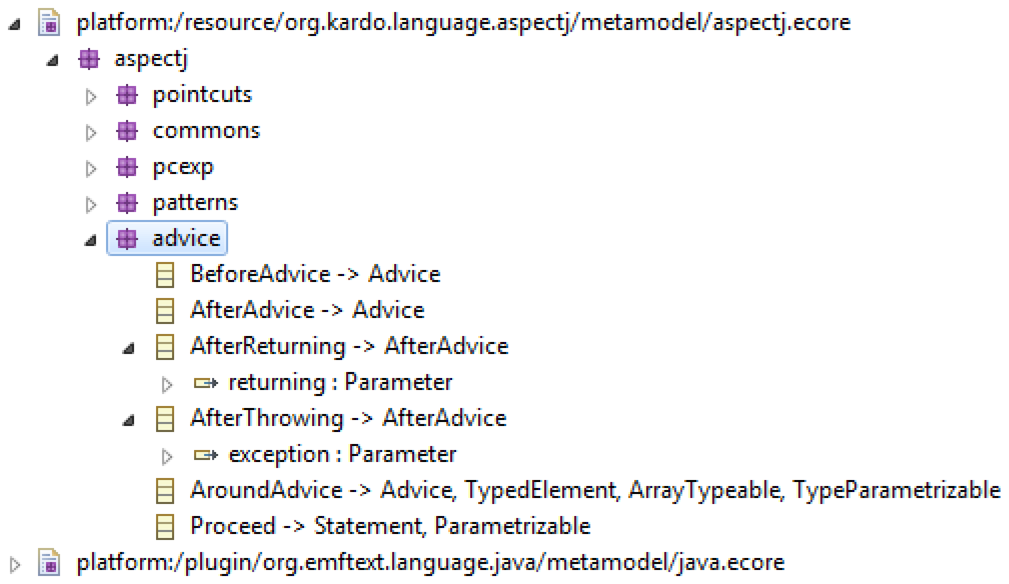
```
platform:/resource/org.kardo.language.aspectj/metamodel/aspectj.ecore
    aspectj
        pointcuts
        commons
        pcexp
            PointcutExpression
                pointcutExpressionChildren : PointcutExpressionChild
            PointcutExpressionChild
            OrPointcutExpression -> PointcutExpressionChild
                orPointcutExpressionChildren : OrPointcutExpressionChild
            OrPointcutExpressionChild -> PointcutExpressionChild
            UnaryPointcutExpression -> OrPointcutExpressionChild
                unaryPointcutExpressionChild : UnaryPointcutExpressionChild
                negated : EBoolean
            UnaryPointcutExpressionChild -> OrPointcutExpressionChild
        patterns
        advice
platform:/plugin/org.emftext.language.java/metamodel/java.ecore
```

Figure 3. The Pointcut Expression Package Structure

The Pcexp package has several classes defining PointcutExpressions and combining them with and (&&) and or (||). The hierarchy of how it is done is through having children, a PointcutExpression has children which are of the type OrExpression or OrExpressionChild, and OrExpression has children of the type OrExpressionChild, AndExpression or AndExpressionChild. AndExpression similarly has children of the type AndExpressionChild.

This structure of the expressions follows the structure used by JaMoPP for Java's expressions, except that we did not make our defined PointcutExpression extend on any of Java's expression classes. A different approach would have been to allow the PointcutExpressions to also inherit from Java's Expression, or even as low level as Java's ConditionalExpression. While this may seem harmless as it allows ConditionalExpressions to be used as PointcutExpressions, it lets PointcutExpressions be used in Java syntax such as statements, which should not be allowed. This is because a PointcutExpression should only be found within either a Pointcut or Advice header, and if written in a statement it could be stored as a variable or be misused in the body of a method while it has a dynamic context.

2.3.4 The Advice Package

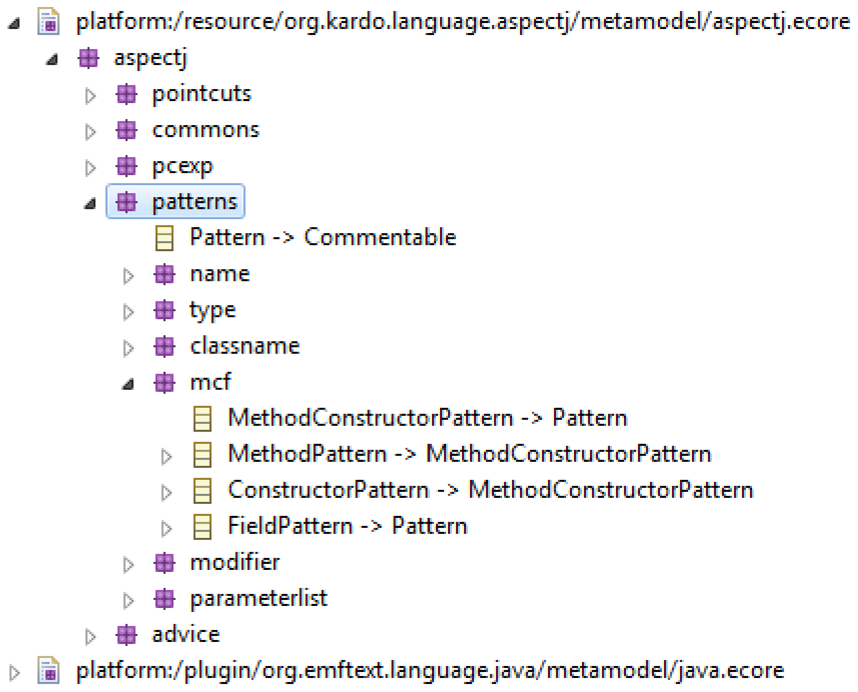


```
platform:/resource/org.kardo.language.aspectj/metamodel/aspectj.ecore
  aspectj
    pointcuts
    commons
    pcexp
    patterns
    advice
      BeforeAdvice -> Advice
      AfterAdvice -> Advice
      AfterReturning -> AfterAdvice
        returning : Parameter
      AfterThrowing -> AfterAdvice
        exception : Parameter
      AroundAdvice -> Advice, TypedElement, ArrayTypeable, TypeParametrizable
      Proceed -> Statement, Parametrizable
platform:/plugin/org.emftext.language.java/metamodel/java.ecore
```

Figure 4. The Advice Package Structure

The Before and After advice are simple and inherit directly from our basic Advice entity without extending it with any additional functionality. The After Returning and After Throwing extend the After advice with the addition of one extra reference to the returned or thrown parameter respectively. Finally, the return type of Around advice determines the need to inherit from *TypedElement. ArrayTypeable* and *TypeParametrizable* allow the return type to be an array or a generic respectfully. We also cover the possibility for a *proceed* statement in the Around advice with the *Proceed* entity.

2.3.5 The Patterns Package



```
platform:/resource/org.kardo.language.aspectj/metamodel/aspectj.ecore
    aspectj
        pointcuts
        commons
        pcexp
        patterns
            Pattern -> Commentable
            name
            type
            classname
            mcf
                MethodConstructorPattern -> Pattern
                MethodPattern -> MethodConstructorPattern
                ConstructorPattern -> MethodConstructorPattern
                FieldPattern -> Pattern
            modifier
            parameterlist
        advice
platform:/plugin/org.emftext.language.java/metamodel/java.ecore
```

Figure 5. The Patterns Package Structure

Due to the numerous symbols and combinations, and possible ambiguity, implementing the pattern matching mechanics proved to be one of the hardest parts of designing the metamodel. We used the abc AspectJ grammar [12] as a reference guide for this particular part of the process.

There are three important families of patterns. The first one is located in the *mcf* package. It contains the *FieldPattern,* the *ConstructorPattern* and the *MethodPattern.* The first one is used in case of a *get* and *set* pointcuts. The second is used in the context of *initialization* and *preinitialization* pointcuts. The final one and the ConstructorPattern are subtypes of the *MethodConstructorPattern* since many of the pointcuts can match either a method call or a constructor call (e.g. *call, execution, withincode).* The second family of patterns match class names. They are located in the *classname* package and are used with respect to the *within*, *handler* and *staticinitialization* pointcuts. The final big family of patterns is responsible for matching types and identifiers and is located in the *type* package. Patterns from this family are used with *this*, *target* and *args* pointcuts.

The rest of the patterns serve as parts or helpers to the patterns from these three families.

2.4 Transformation of AspectJ models to Java (JaMoPP) models

This section' s main contributor was Hristofor Mirchev. In the practical work we divided it a bit more equally with myself working on the transformations for Advices - not the Proceed statement however, and Hristofor on the remainder of the transformation rules.

Aspectj 5 introduced an alternative to writing aspect declaration in the old traditional way by incorporating an annotation-based style. The gained benefit was that now programs written in that style can be compiled with a regular Java5 compiler and separately be woven at a later stage. Typically both of these processes are encapsulated in the ajc compiler. Separating these two concerns (compiling and weaving) is what allows us to focus on making a moular AspectJ implementation and handle the compiling and weaving processes with traditional tools (i.e. the Java compiler javac and AspectJ' s load-time weaver). Unlike in other AspectJ compilers like abc, not having to implement complex weaving mechanics is the key factor for an easier and faster extension development.

The following table provides an example overview of the main Aspectj concepts written in both regular and annotation-based styles, taken from the official *AspectJ 5 Development Kit Developer' s Notebook* [1].

Regular style	Annotation style
public aspect <Foo> { }	@Aspect **public class** <Foo> { }
pointcut <AnyCall>() : <**call**(* *.* (..))>;	@Pointcut("<call(* *.* (..))>") **void** <AnyCall> () { }
before() : <**call**(* *.*(..))> { }	@Before("<call(* *.*(..))>") **public void** bfAdvice () { }
after() : <**call**(* *.*(..))> { }	@After("<call(* *.*(..))>") **public void** afAdvice () { }

after() **returning**(Foo <f>) : <**call**(* *.*(..))> { }	@AfterReturning(**pointcut**="<call(* *.*(..))>", **returning**="<f>") **public void** afrAdvice (Foo <f>) { }
after() **throwing**(Exception <e>) : <**call**(* *.*(..))> { }	@AfterThrowing(**pointcut**="<call(* *.*(..))>", **throwing**="<e>") **public void** aftAdvice (Exception <e>) { }
Object around(int <i>): <setAge(i)> { **return proceed**(); }	@Around("<setAge(i)>") **public Object** arAdvice (ProceedingJoinPoint jp, int <i>) { **return** jp.proceed(); }

Table 1. AspectJ Components in Regular and Annotation Style

To get the model we obtain after parsing from the regular AspectJ style to the annotation style, we must use a model transformation language. More precisely, we need a model-to-model transformation language to translate a model conforming to our AspectJ metamodel to one that conforms to the Java metamodel provided by JaMoPP. We considered two of the most popular and mature transformation frameworks that support EMF-based models ‑ *Model-to-Model Transformation* (MMT)[13] and *Epsilon* [14].

MMT consists of two very distinct model-to-model toolkits ‑ *QVT* and *ATL*. QVT is a standardized set of three model-to-model languages ‑ QVT-Operational, QVT-Relations and QVT-Core. The first one is an imperative language, while the other two are both declarative and are therefore commonly jointly called QVT-Declarative [15].

ATL was initially designed as an alternative to QVT before getting paired with it in MMT. ATL supports both imperative and declarative styles of writing transformations. The recommended style is declarative as it is better for simple and straightforward transformation rules, but imperative can also be used for more complex ones [16].

Epsilon is a rich toolset that can be used for model validity, model comparison, code generation and model-to-model transformation. The framework provides a language for each of those functionalities. All of those languages however are minimal extensions built on top of a common imperative language ‑ the *Epsilon Object Language* (EOL). The language for the model-to-model transformation is called *Epsilon Transformation Language* (ETL). Like ATL it is a hybrid language in the sense that it supports both imperative and declarative styles of writing [14].

In comparison, both MMT and Epsilon allow us to have rules that transform any number of input models to any number of output ones. Both frameworks also support imperative and declarative styles of writing. Finally, both frameworks are mature and rich, and offer a diverse set of extra functionality like syntax highlighting, error detection and debugging in Eclipse. With respect to these common classification criteria Epsilon and MMT are similar to each other. The only deciding requirement we had was how easy it is to call class methods outside the context of the transformation, since, as can be seen from Table, to transform a pointcut or advice a programmer would need to transform the pointcut expression they contain to a string and pass it along as a parameter of the annotation. Since no transformation language would be able to perform this task natively, we needed an easy way to call our generated printer to do that.

In QVT such cases are referred to as “black box operations” [17]. We found this approach to make the project and its design more complicated. Epsilon, on the other hand, has a clear way of doing this and even lists it among the main features to use the framework [14]. Later in this section we are going to explain more in-depth exactly how we accomplished this task as it proved to be rather challenging, but this was the sole reason we picked Epsilon over MMT. The choice, however, will also allow us to re-use code from our transformation rules to implement model validity or unit tests for the transformation in future work, as all languages in Epsilon share a common syntactical foundation.

ETL transformations are organised in a module that can contain an arbitrary number of uniquely named transformation rules. As well as transformation rules, an ETL module can optionally contain any number of *pre* or *post* blocks of statements which are executed before or after the transformation respectfully. The following listing displays the syntax for a transformation rule and the post/pre blocks.

```
(pre | post) <name> {
   statements+
}

(@abstract)?
(@lazy)?
(@primary)?
rule <name>
   transform <sourceParameterName> : <sourceParameterType>
   (, <sourceParameterName> : <sourceParameterType>)*
   to <rightParameterName> : <rightParameterType>
   (, <rightParameterName> : <rightParameterType>)*
   (extends <ruleName> (, <ruleName>)*)? {

   (guard (:expression) | ( { statementBlock }))?

   statements+
}
```

Listing 1. Syntax of a Transformation Rule and Pre/Post block

The pre and post blocks consist of the respective identifiers (pre or post), an optional name for the block and the set of statements to be executed. The transformation rule can be declared as abstract, lazy or primary via annotations, followed by the rule identifier and the rule name. The source and target models are declared following the transform and to keywords. A rule can also extend any number of different transformation rules declared after the ‘extends’ keyword. Apart from the EOL statements a programmer can also specify a guard statement to limit the applicability of the rule to a selected subset of source models [18].

In the following subsections we are only going to demonstrate our transformation rule for the Before advice as the rest are analogous. We believe it still sufficiently captures most of the challenging logic we faced when designing the transformation. As can be seen from Table, a Before advice declaration using the annotation style is just a regular public Java method that returns *void* and has the *@Before* annotation. Two important considerations here are:

1. Although the advice declaration does not have a name, the Java method should have an unique name.
2. The pointcut expression that the advice contains is passed as a string parameter to the annotation.

Listing 2 shows our transformation rule for the Before advice.

```
1.   rule Advice2Method
2.      transform ajAdvice : aspectj!Advice
3.       to jMethod : java!ClassMethod {
4.
5.      jMethod.name = getUniqueAdviceName(ajAdvice);
6.      jMethod.parameters = ajAdvice.parameters;
7.   }
8.
9.   rule BeforeAdvice2Method
10.     transform ajBeforeAdvice : aspectj!BeforeAdvice
11.      to jMethod : java!ClassMethod
12.      extends Advice2Method {
13.
14.      jMethod.annotationsAndModifiers.add(getAnnotation(ajLibBefore!  Commentable.allInstances().first(),
     "Before", ajBeforeAdvice.pcref));
15.      jMethod.annotationsAndModifiers.add(new java!Public);
16.
17.      jMethod.typeReference = new java!Void;
18.      jMethod.statements = ajBeforeAdvice.statements;
19.  }
```

Listing 2. Before Advice Transformation Rule

Lines 1 – 7 describe a common rule that all other advice rules extend. It takes care of the first consideration we noted by calling a *getUniqueAdviceName* helper method and passes along the parameters that the advice might have as parameters of the new method. Lines 15, 17, 18 set the method to be public, return void and pass along the body of statements the advice contains as the body of the new method. These three lines are common for all advice types except *Around* (the annotated method for Around has the return type of the Around advice itself and a *proceed* statement will get transformed rather than copied verbatim). Thus, one can argue that those three lines could also be extracted to the common advice rule and have a separate and specific rule for Around. We liked our approach better as an Around advice is still a type of advice and thus the relationship is still of an *is-a* kind, which is represented by inheritance. Finally, line 14 and the call to *getAnnotation* handle the second consideration we mentioned.

Let us now demonstrate the implementation of the two methods: getUniqueAdviceName and getAnnotation.

```
1.  pre {
2.      var globalAdviceCounter : Integer = 0;
3.  }
4.
5.  operation Any getUniqueAdviceName(ajAdvice : aspectj!Advice) : String {
6.      globalAdviceCounter = globalAdviceCounter + 1;
7.      var name : String = "aj$" + ajAdvice.eClass.name + "$" +  ajAdvice.eContainer.name + "$" +
    globalAdviceCounter + "$" +          ajAdvice.hashCode();
8.
9.      return name.replace(" ", "_");
10. }
```

Listing 3. Getting the Unique Name for Advice Methods

Listing 3 shows how we handle the first consideration of generating an unique name for the advice method. We looked at how the official AspectJ compiler ajc handled the same issue and tried to imitate the same behavior. Line 7 shows how we form the name of the method. We start with the string *aj (*a mnemonic for AspectJ), concatenate the type of advice (in our example this would be “Before”), the name of the aspect that contains the advice, a global counter of advice and finally we add a hash value. We use a pre block to create an advice counter variable as since it gets executed only once before the actual transformation, it simulates the global variable we need.

```
operation Any printModelElement(elem : Any) : String {
    var resourcePrinter = new Native("org.kardo.language.aspectj.resource.aspectj.mopp.ParameterlessAspectjPrinter");

    if (elem.isDefined()) {
        return resourcePrinter.printElement(elem);
    } else {
        return "";
    }
}
…
operation Any getAnnotation(libModel : Any, annotation : String, pointcutExpression : aspectj!PointcutExpression) :
java!AnnotationInstance {
    var anno = makeAnnotationInstance(libModel, annotation);

    if (pointcutExpression.isDefined()) {
        var pcExp : String = printModelElement(pointcutExpression);
        var param = new java!SingleAnnotationParameter;
        param.value = getParameterValue(pcExp);
        anno.parameter = param;
    }
    return anno;
}
```

Listing 4. Getting the Advice Annotation with the Pointcut Expression

Creating and setting up the annotations was one of the major issues we encountered while working on the transformation. We were facing two problems with this task.

First, how to create an instance of the annotation entity of the Java metamodel we use that “points” to the actual annotation located in the AspectJ library. An annotation in JaMoPP’s Java metamodel is represented by an optional namespace part (i.e. an ordered set of strings) for the fully qualified annotation name, and a name part (i.e. an element of type *Classifier*). Classifier is another entity in the metamodel, but it is abstract, which means we can not directly “create” it and set it to the correct annotation name (i.e. Before in our case). To solve the problem we took all the actual .java files of the annotations from the AspectJ library, parsed them with JaMoPP and fed the resulting models to the transformaton. At that point, setting the name part of the annotation we are trying to create was just a mapping to the correct model. The query to obtain the correct model is done in the *makeAnnotationInstance* method.

The second problem we faced was how to get a string representation of the pointcut expression that advice contain. Lines 1–9 of Listing 4 demonstrate our solution. The easiest way to accomplish this task is to call the generated printer for our AspectJ model and pass the pointcut expression. ETL offers an easy way to access classes outside of the transformation context, however the only requirement is that the class has a no-argument constructor. Since the generated printer class does not have such a constructor, we decided to write a custom printer class that extends it and has such a constructor. We prefer this over the alternative of simply adding such a constructor in the generated printer, as it is more extensible and imposes a clear separation of the generated from the non-generated code. On line 2 we create a variable that points to the custom printer and on line 5 we call its *printElement* method to obtain the string.

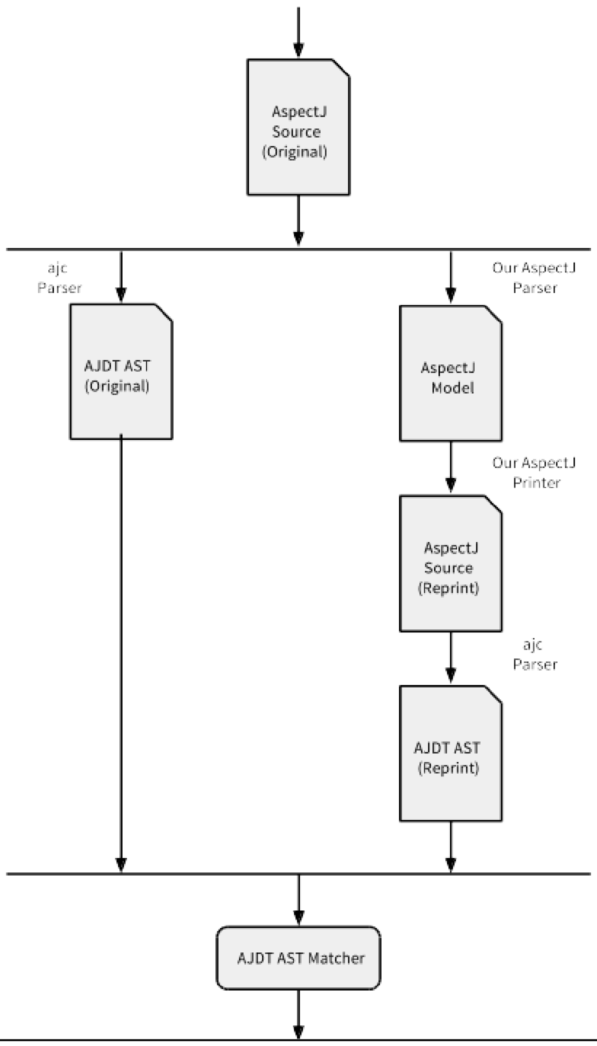
2.5 Testing and Evaluation

This section’ s main contributor was Hristofor Mirchev. In the practical work this was split more finely with myself writing the custom test cases, and Hristofor Mirchev working on setting up the testing environment.

In the previous subsections we presented our AspectJ metamodel as well as some challenges and design decisions we faced along the way. In order to evaluate the approach we have devised a test suite to demonstrate the correct parsing of AspectJ applications in reference to the official AspectJ compiler - ajc.

The goals of the suite are to test 1) that our parser accepts valid AspectJ code, 2) that the model instance created after parsing has the expected structure, 3) that the generated printer outputs a code representation of the model that is semantically equivalent to the input. To achieve these goals we employed a testing process similar to how JaMoPP was tested [7]. The process is shown on Fig. 6.

Figure 6. Test Process for the AspectJ Implementation



Beginning with a valid AspectJ source file, both our generated parser and the reference one (the ajc) process the file and create their respective internal representations of it. In our case, this is a model instance of the AspectJ metamodel with unresolved cross-references. In ajc’ s case, this is an abstract syntax tree (AST). Next, our generated resolvers attempt to resolve the cross-references after which the generated printer reprints the model in its text form. The reprinted source file is fed to the reference compiler which creates another AST. Finally, the AST of the reprint is compared to the original one via an AST matcher provided by ajc.

We believe this approach meets the goals we set out to achieve. First, if our parser accepts all the valid AspectJ test programs we give it and not throw a parsing error, then we can conclude that the soundness property in our first goal is met. Secondly, the structure of the model instance is checked for correctness by the AST matcher. If there are any unresolved or missing elements, they will cause resolving errors and not get reprinted, which will be detected by the AST matcher. Lastly, any other mismatches that might raise error messages will also be detected by the matcher. In those cases, we manually checked the reprinted source file and compared it with the original to discover the source of the error. In nearly half of our test cases we exhibited such reprinting errors although all of them were parsed without errors. Often times white space, empty blocks and other layout information led to the mismatches causing the errors.

Due to limitations we imposed on the metamodel by design, running our test suite with official AspectJ benchmarks was impractical as it would require us to go through each source file and modify uses of functionality we did not implement. Thus, as input for the test process we provided 18 AspectJ files we custom wrote ourselves. We tried to achieve maximum coverage by exposing every variation of each AspectJ element in our model. The overview of the test files separated over the packages in our metamodel is the following:

- advice package – Contains 5 test files, one for each advice type. Each file tests for an empty advice, for an advice that exposes a parameter, for a *strictfp* advice and for an advice written in annotation style.
- commons package – Contains 3 test files. The first one test for all possible classifier declarations in a compilation unit (i.e. an aspect, a class, an annotation, an enum and an interface). The second tests for all possible non-AOP contents of an aspect (i.e. a nested class, a field and a regular method). The last one tests for all possible pointcut declaration variations (i.e. one without a modifier, one without a pointcut expression, one where the pointcut expression is just a primitive pointcut and one where it is a conjunction).
- patterns package – Contains 8 test files that also thoroughly test all the variations of the pattern types we have modeled (refer to the patterns subsection in Sec. 0).
- pcexp package – Contains 1 test file. It checks for a pointcut declaration with a regular pointcut expression and one with a negated pointcut expression.
- pointcuts package – Contains 1 test file. The file contains 18 pointcut declarations that test out

all possible primitive pointcuts.

The testing was automated with the Junit framework and special effort went into making the environment easily extendable with more tests. A programmer simply has to put a valid AspectJ source file ending with the .aspectj extension of our implementation in the src-input folder of the testing project and then rerun the tests. In the end all 18 files passed the test suite.

In conclusion, we can say that while this test process does not guarantee completeness, it does give us enough confidence that our AspectJ implementation is sound and can be the foundation of language extensions that can be used in practice.

To incorporate a similar testing process for the transformation, we would first have to write a script to run all the test files we have through our parser. After that, we would execute the transformation on the resulting models and run them through the JaMoPP printer. The resulting Java source files can then be put through the same testing process. The difference being that this time we use JaMoPP’ s parser, the standard Java compiler javac and the JDT AST matcher rather than their respective AspectJ equivalents. Setting up such a testing environment would take much time in integrating JaMoPP, javac and the JDT matcher, so we left this for future work and opted for a smaller and more manual approach.

We fed the 18 test files we had made to our parser and ran the transformation on the resulting models. After that, we only inspected the new models that the transformation produced by hand. The results satisfied our expectations and therefore, we can say with some amount of certainty that the transformation, just like the AspectJ implementation, is sound.

3. Part III

3.1 Introduction

In this section I will be presenting my Instance Pointcuts language implementation, as well as applying the language to some case studies to analyse and compare the language to plain Java. Following the design process for the AspectJ implementation, I used the same structure of working by first implementing the metamodel for the extension language using MDE techniques. Once that was complete I then transformed the result to our defined AspectJ language with Epsilon's Transformation Language for model-model transformation.

Since it is not the main goal of this thesis to guarantee correctness of the language beyond that the language works, the entities behave as expected in my case study, and that the syntax rules are correct as Kardelen's research has specified [24]; the testing of the metamodel itself has been left out. This may make it uncertain if the language correctly enforces all of its syntax rules, and also if multisets behave correctly with all types of objects, especially when a composite instance pointcut is created as a union or intersection of multiple multisets. For the purpose of answering my research questions however it is not vital to test the limits of the language's capability. Instead, our main goal is to implement case studies in our language, which will automatically test the functionality of Instance Pointcuts; if the application still behaves as expected in runtime when written in our language, then we can already safely assume the language works.

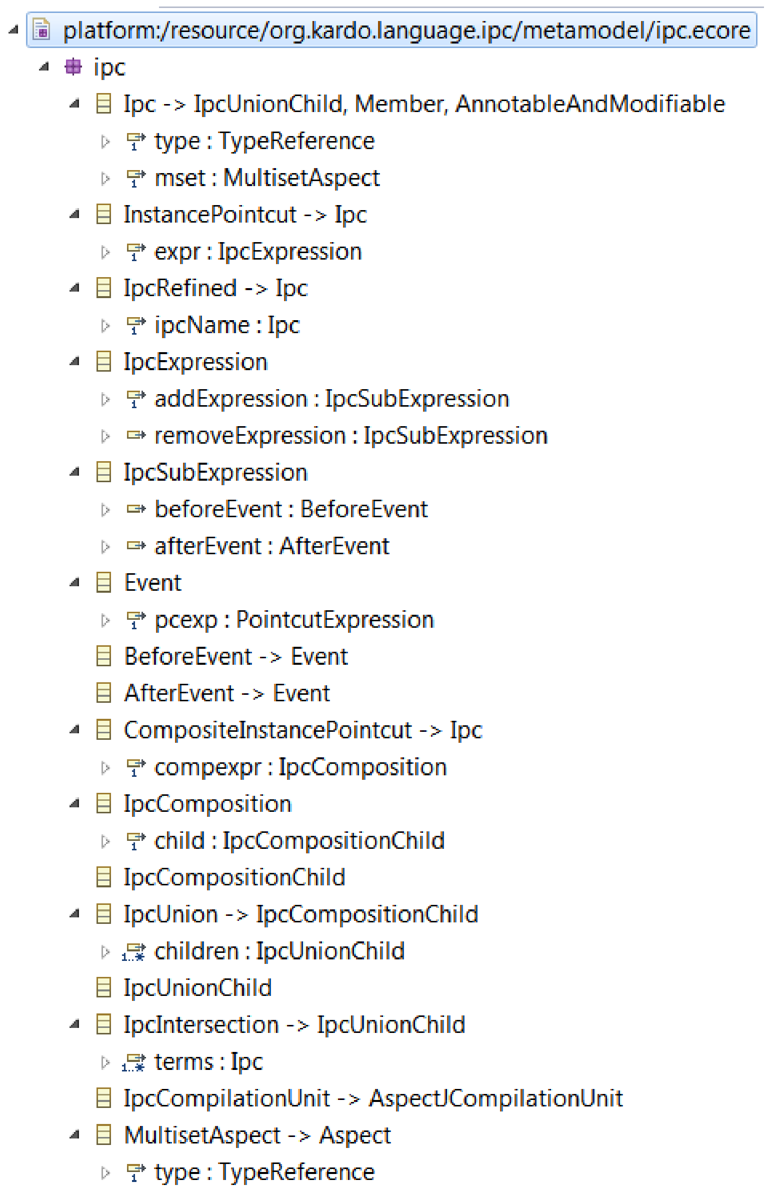
3.2 Instance Pointcuts Implementation

The strategy I decided on for implementing the language structure was to keep it simple, and only contain the entities necessary for the features I wanted the language to support. These features were selected from the specification of the language defined by Kardelen's research [24]. From that list I first chose the features that were either an integral part of the language, and then added features based on the complexity to implement, both in the metamodel and to transform to AspectJ, and what impact the feature would have when included in a case study application. Due to time constraints not all features could be implemented, but this does not have a major impact on the usefulness of the language. As the number of classes for instance pointcuts is very small compared to those of the AspectJ implementation, no further structuring was needed, as it would add unnecessary complexity to the metamodel, so all classes were placed in the same package.

The language can be broken down into several features:

1. Instance Pointcuts; the main feature of the language which contains instance pointcut expressions, multisets and a typereference.
2. Instance Pointcut Expressions; these expressions are made up of a mandatory sub expression for adding objects, and an optional sub expression to remove objects from the multiset. Sub expressions contain the events that need to occur within an object to be added or removed to the multiset.
3. Events; these are made up of before and after events, similar to AspectJs advices.
4. Composite Instance Pointcuts; these add the first level of complexity to the language, as they allow new instance pointcuts to be composed of existing ones.
5. Instance Pointcut Refinement; these special type of instance pointcuts allow existing ones to be refined with varying levels of granularity, each level adding higher complexity to the language.

6. Multisets; these get generated with each instance pointcut to store all the objects that the events find, and is accessible to users in an unmodifiable set only.



```
platform:/resource/org.kardo.language.ipc/metamodel/ipc.ecore
  ipc
    Ipc -> IpcUnionChild, Member, AnnotableAndModifiable
      type : TypeReference
      mset : MultisetAspect
    InstancePointcut -> Ipc
      expr : IpcExpression
    IpcRefined -> Ipc
      ipcName : Ipc
    IpcExpression
      addExpression : IpcSubExpression
      removeExpression : IpcSubExpression
    IpcSubExpression
      beforeEvent : BeforeEvent
      afterEvent : AfterEvent
    Event
      pcexp : PointcutExpression
    BeforeEvent -> Event
    AfterEvent -> Event
    CompositeInstancePointcut -> Ipc
      compexpr : IpcComposition
    IpcComposition
      child : IpcCompositionChild
    IpcCompositionChild
    IpcUnion -> IpcCompositionChild
      children : IpcUnionChild
    IpcUnionChild
    IpcIntersection -> IpcUnionChild
      terms : Ipc
    IpcCompilationUnit -> AspectJCompilationUnit
    MultisetAspect -> Aspect
      type : TypeReference
```

Figure 7. The Complete Structure of the Instance Pointcuts Metamodel

3.2.1 Instance Pointcuts

As Figure 7 shows, all the types of instance pointcuts inherit from an abstract parent named Ipc, which should not be confused with the child Instance Pointcut. The Ipc entity stores the *TypeReference* and Multiset for each type of instance pointcut, and it also extends Java's *Member* and *AnnotableAndModifiable* classes, which allow it to be placed within Java classes as well as aspects similar to AspectJ's *Pointcut*. This also means that each Ipc type has a name, which is required for reusability and composition; hence it extends the IpcUnionChild, as each Ipc can be used in a composite instance pointcut expression.

While Ipcs are able to store individual objects based on more than just their class type – they do also store a *TypeReference* themselves, which makes it possible to put a limit on the scope of objects these pointcuts look at within an application. Both the name and type of the Ipc is mandatory.

The child entity Instance Pointcut stores a mandatory instance pointcut expression, or IpcExpression, alongside the fields specified by the Ipc entity. The reason the instance pointcut expression is not stored in the Ipc as well is because refined instance pointcuts (at the level of granularity I have implemented) and composite instance pointcuts do not need to store them. Since refined and composite instance pointcuts are made up of a reference to existing Ipcs; there is no need to store the expressions as well since these pointcuts will not modify the expression in any way. While composite instance pointcuts do store an expression themselves, it is of a different type, which is discussed later.

3.2.2 Instance Pointcut Expression

Instance Pointcut Expressions contain up to two subexpressions, and the instance pointcut expression itself does not need to inherit any properties from AspectJ or Java since it behaves differently to a Pointcut expression or a Java expression.

3.2.3 Events

Events, while similar to AspectJ's Advices in terms of being an abstract entity that Before/After events inherit from, does have some fundamental differences that prevent them from inheriting from Advice. Events do contain a pointcut expression, which could be a named or composed pointcut. But unlike Advices, Events do not allow a user to apply any adaptation or modification of the object that is returned by the pointcut expression before or after it is passed to the multiset. Also as Events should not be definable outside the scope of an instance pointcut expression, it was best to make Event have no dependency on Advice at all.

3.2.4 Composite Instance Pointcuts

Composite Instance Pointcuts inherit from the abstract Ipc class, and contain a special type of expression named an IpcComposition, which allow multiple existing instance pointcuts to be composed as a new single instance pointcut. The composition can happen in several ways; it can either be composed of a single Ipc, or a union of two or more Ipcs, or the intersection between two or more Ipcs, or both a union and intersection. I use the class Ipcs for the composition, because the composition may be of either instance pointcuts or refined instance pointcuts, or an existing composition – allowing for nesting of instance pointcuts.

The union and intersection keywords of the composition are important since they refer directly to the multiset that each Ipc holds, and will mean if either the union or intersection of these sets of objects is taken for the composition.

3.2.5 Refined Instance Pointcuts

While the refinement of instance pointcuts is a feature of the language that can become quite large and complex, I decided to implement it only in its lowest level of granularity, which enables users to define a refined instance pointcut, with a reference to the existing Ipc it is refining from. At this level of granularity, it only means it can specify a new name and type for the Ipc. This already is enough for a basis which can be extended upon if a feature-complete implementation of the language is desired.

3.2.6 Multisets

Multisets, as a class feature of the language does seem out of place, since unlike the other class features of the language it does not provide any aspect-oriented functionality, but it still has an important purpose to store objects, or sets of objects within one multiset. It provides its functionality for adding and removing objects to its set in the form of an Aspect, which is able to monitor the Events of an Instance Pointcut and whether the event belongs to an Add or Remove subexpression to determine what action it should do to the object.

One Multiset should be generated for each Ipc, with the same name and type of the Ipc so it knows to which Ipc is belongs to. The metamodel however only encompasses the name and type the multiset will have; the generated code is not generated until the transformation into AspectJ. This was fundamentally done due to desiring a simple and clean metamodel; and having the complete aspect syntax be generated at this stage would significantly increase the complexity of the language. Since compiling and executing of any Instance Pointcut application would be done through transformation into AspectJ and Java, this decision has no negative consequences.

3.3 Instance Pointcuts Transformation

The implementation of the metamodel resulted in the instance pointcut language supporting most of the important features of the specified language description – enough for applications in Java to at least be supported to a degree where some comparisons can be drawn been an Instance Pointcut implementation and a Java implementation. As far as this thesis is concerned, the metamodel of the language is at least complete and the transformation could be implemented, using the Epsilon Transformation Language as we did with AspectJ.

There are a number of entities that the Instance Pointcut language defines, but only few of them have an easy mapping to an AspectJ element. Unlike with the AspectJ language, we do need to specify how each property matches to either an AspectJ or Java element, as we cannot in this case simply transform properties to annotated form. This is because there is no existing compiler that can take annotated Instance Pointcuts code and compile it to a form that can be executed in a Java environment, and we do not plan on defining one.

The mapping of Instance Pointcut entities to AspectJ entities is as follows:

Instance Pointcut	AspectJ / Java
IpcCompilationUnit	AspectJCompilationUnit
Instance Pointcut, Refined/Composite	Aspect
IpcSubExpression	Java Member
Before Event	Before Advice

After Event	After Advice
Multiset	Aspect
IpcComposition	Java Member

Table 2. Transformation of entities from Instance Pointcuts to AspectJ/Java

As the table shows, some transformations are straightforward, such as events to advice, and from one compilation unit to the other. It also highlights how the AspectJ application will be structured once the Instance Pointcut application gets transformed into it.

While instance pointcuts are a special type of pointcut, they do not get transformed to a pointcut entity for several reasons. The syntax for instance pointcuts is structured very differently to ordinary pointcuts, and it does not take a *PointcutExpression*. Instead, as it contains an *IpcExpression*, which holds an add/remove expression, which can each have a before or after event or both; it ends up holding a lot more data than an ordinary pointcut. For this reason, transforming instance pointcuts to Aspects enables it to contain all the sub expressions and events as Java Members.

Refined instance pointcuts at the low level of granularity I have implemented do transform into mostly empty Aspects, but an Aspect is still the best match for this entity, since it is a type of instance pointcut, but one that simply points to an existing instance pointcut. For future extensions if a higher level of granularity is achieved for refinement; the Aspect will in turn become larger in size as it will contain the refined instance pointcut expression or even refined events.

Finally, composite instance pointcuts again transform into Aspects, but have a higher level of complexity than refined instance pointcuts. While it is easily possible to get all the Ipc entities that a composite instance pointcut is made up of, and then add them all as Java Members to the composite aspect, this does have a negative consequence. The consequence is that the way the instance pointcut is composed gets lost in transformation, so the Ipc entities are all loosely placed within the aspect, without a reference to whether their multiset should be created as a union or an intersection. Since the metamodel definition does not include the logic of how multiset union or intersection works or should occur, it is up to the transformation stage to define this logic. Since composite instance pointcuts are not a fundamental part of the language, and only a useful feature in certain scenarios, I have left this for future work.

3.3.1 Multiset Transformation

As discussed previously, Multisets in Instance Pointcuts are only defined as an entity that holds a name and type variable, and in this transformation stage it gets properly generated. The generating of the Multiset Aspect contents is largely hardcoded, since the functionality is the same between Multisets apart from the naming of methods, pointcuts, advices, and the Aspect name. To avoid having an equivalent name to the instance pointcut Aspect it belongs to, it has the literal ‘Multiset’ attached to its name. Along with behaving as an Aspect, it also has one important method defined which returns an unmodifiable state of the set to the user which represents all current objects which during runtime have met the event criteria to be added to the set, and not yet matched an event to be removed.

The Multiset Aspect code that should be generated for each Ipc would resemble the following structure:

```
import java.util.Collections;
import java.util.Set;

import com.google.common.collect.HashMultiset;
import com.google.common.collect.Multiset;
import com.google.common.collect.Multiset.Entry;

public aspect MultisetAspect {

    protected Multiset<Object> nameHS = HashMultiset.create();

    public Set<Entry<Object>> getHSet() {
        final Set<Entry<Object>> tempSet = Collections.unmodifiableSet(nameHS.entrySet());
        return tempSet;
    }

    // after add
    after(Type instance) : call(* ..) && target(instance) {
        addToNameHS(instance);
    }

      // before add
      before(Type instance) : .. target(instance) {
        addToNameHS(instance);
    }

    private void addToNameHS(Type instance) {
        if (!nameHS.contains(instance)) {
            nameHS.add(instance);
            // we add for the first time
            addedFirstToNameHS(instance);
        }
        else {
            nameHS.add(instance);
        }
    }

    pointcut addedFirstToNameHS(Type instance) : call(addedFirstToNameHS(..)) && args(instance);

    private void addedFirstToNameHS(Type instance) {}

    private void removeFromNameHS(Type instance) {
        if (!nameHS.contains(instance)) {
            //error
        }
        else {
            nameHS.remove(instance);
        }
    }
}
```

Figure 8. The Multiset Aspect code

3.4 Case Studies

3.4.1 Bohnanza

Bohnanza is a card game for two or more players in which the objective is to get the most coins from growing and harvesting beans. There are 11 different types of beans, and players can only grow one type of bean in a field at a time. Players can have multiple fields however, and each player has a hand of cards which are the beans to play in their field. Like most card games, there is a deck of cards which players draw from each turn, as well as a discard pile of cards which contains cards that resulted in no coin turnover when harvested.

The specific bean-card details, rules, and other entities such as the user interface, the treasury, or the trading area of the game are not important for this case study, but it is important to know that the card game is extendable, with a number of different variations in terms of both rules, structure, and entities that are present in the game.

The Bohnanza game was group project I implemented with Roland Balk [25], before being aware or having any experience or expertise in the topic of Instance Pointcuts, only with knowledge of Aspect-Oriented techniques. The application was also made with little care for scattered code or crosscutting concerns; only good quality Java code and its standards, so it would be an ideal candidate as a case study for being implemented in Instance Pointcuts.

The benefits of using one case study as my own previous work are that I am fully aware and am understanding of the original code and the purpose each entity has for the application. Also knowing what some of the strengths and drawbacks of this implementation means I can do a fairer comparison at the end on the resulting implementations.

3.4.1.1 The Java Implementation

For the initial or basic design of the game, we decided to start off with a Game class in which all components of the basic game would be included, and which also initialises the game [25]. This includes allowing a user to choose the number of Players that will be playing as well what type of game will be played (basic or High Bohn). The GUI is then also constructed once all components have been initialised.

As we wanted the design of our game to be both flexible and stable, we wanted to take a very modular approach to the design of all the components of the game, which would also make it much easier to achieve correctness [25]. With this approach, we decided to split each part of the game into their own class, and keep the number of dependencies and relations to a minimum. Each class, such as the Field would then only be concerned with its own role of allowing Beans to be planted and harvested. Complete dependency removal of classes from the Deck or BeanCard classes was not achieved however, as it would have made individual classes more complex if they had to handle card interactions themselves.

Several of the game's classes have little to no relevance to us from an Instance Pointcut point of view, but our main focus is set to improve the way the cards are being used and stored. We particularly aim to improve the interactions that are done from other classes to the deck and other lists storing cards, as several classes have got a number of ways in which they interact with them. If these interactions were to be reviewed and refactored in our Instance Pointcut implementation, we would hope to simplify the structure of the application, by both reducing the amount of code scattering, as well as overall reducing the lines of code of the application. This may not actually improve the performance at all, but it should improve maintainability.

While in this implementation it does show how the structure can support an extension; any extension that were to modify the deck would not be easily done without significant structural changes. The structure of the game as it stands is as follows:

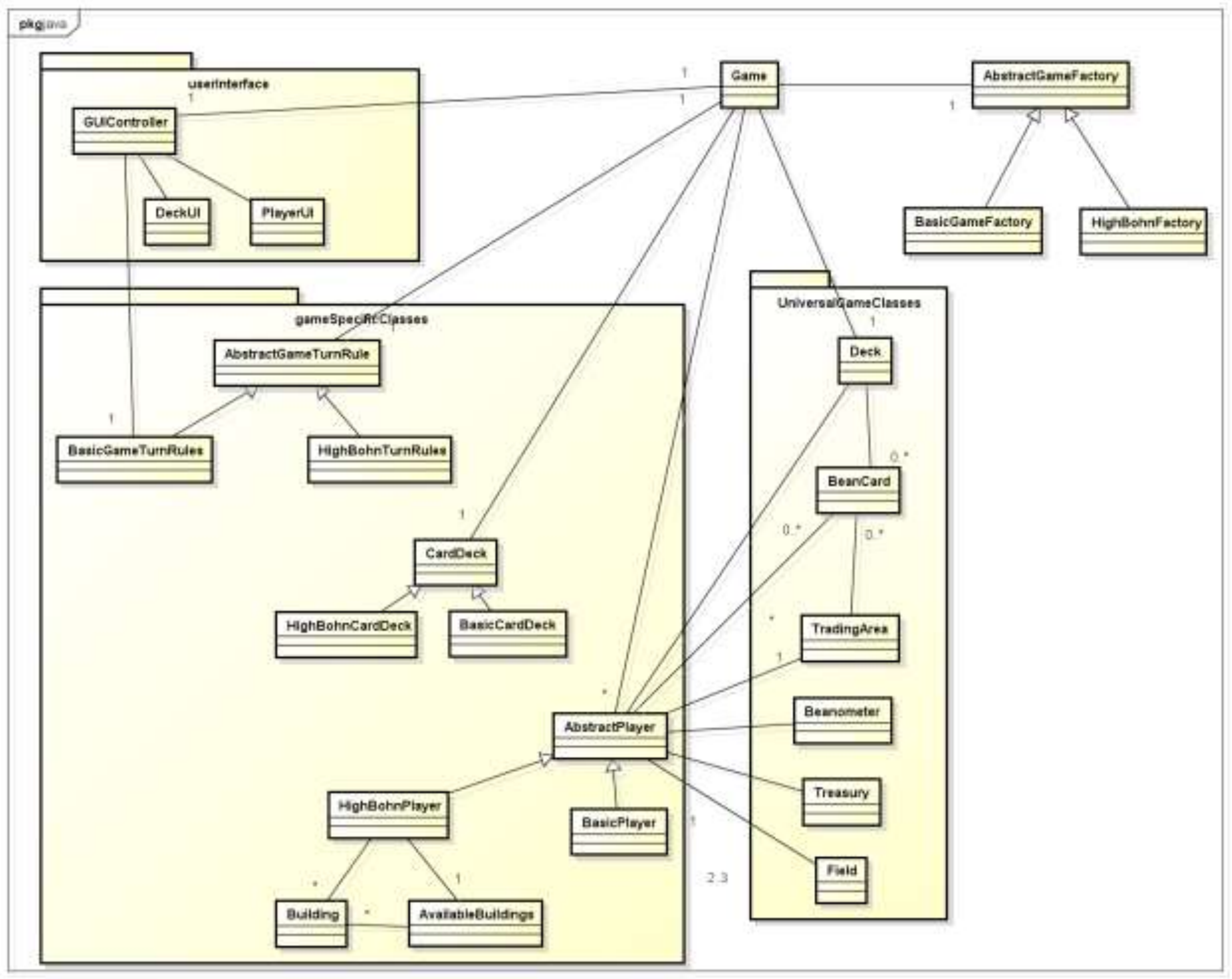


Figure 9. The Bohnanza Application Structure

As figure 9 shows, the complete structure is fairly large and complex, with significant amounts of dependency relations between classes. While my goal is not to improve each structural entity of the application, since we know that Instance Pointcuts is only applicable to certain scenarios, I have focussed my efforts on improving the interaction with the BeanCard class. There are several classes which store a list of BeanCards; the Player, the Field, the Treasury, and the Deck of course.

The Deck class stores two sets of cards actually, not only the deck of cards that is played with, but it also contains the discard pile for the game. This is because the discard pile gets fed back into the main deck once the deck runs out of cards to deal out. The Deck class is noticeably larger due to the need for methods to access, not only get but set, the cards of both sets. Furthermore, the set of cards that deals out cards to players is of the type *LinkedList*, while the discard pile is an *ArrayList*, which adds unnecessary confusion to the class. The Deck class currently consists of 180 lines of code, with 8 methods including the constructor, and 4 of them are accessor methods.

The Player class is divided between a BasicPlayer and a HighBohnPlayer, both with a common parent AbstractPlayer. They both handle their cards similarly; the HighBohnPlayer introduces extra functionality in terms of Buildings, but this does not have any effect on how cards are handled. To properly support any other extension in which a new type of Player is introduced that may handle cards differently, we can't rely on this parent to decide what interacts can be done with cards.

This means that child class Players need to decide this functionality, which did cause some code duplication between the two types of Players. While the hierarchy of players could have been improved to have a new parent type Player that has implemented the card interaction, this was not done here. I can with Instance Pointcuts at least get rid of any current and possible future code duplication in this case by using a pointcut to deal with the cards.

The remainder of the implementation is straightforward, with only the Field and Treasury classes also storing cards and therefore can be improved, but in these cases it will likely only have a small impact on improving code quality.

3.4.1.2 The Instance Pointcut Implementation

The first step of improving the Java implementation for Bohnanza with Instance Pointcuts is to define the textual syntax. I have already identified now which classes I aim to improve, and from that can define the instance pointcuts to replace the way the bean cards are managed. While I do aim to remove several redundant methods from the original Java implementation once the instance pointcuts are in place; I will still be keeping some methods in as they are useful for Pointcut Expressions to match on.

The textual syntax I defined for the whole application is as follows:

```
package gamePackage.universalGameClasses;

aspect CardManager<BeanCard> {

instance pointcut deck<BeanCard> :
      afterevent(call(*.createCardDeck()) ||
        call(*.resetDeck()))
  UNTIL
      afterevent(call(Game.initCards(..)) ||
        call(Game.dealCards(..)))

instance pointcut discardPile<BeanCard> :
      afterevent(call(*.harvest()) && call(*.discardCards(beans)) && target(beans))
  UNTIL
      afterevent(call(*.resetDeck()))

instance pointcut playerCards<BeanCard> :
      afterevent(call(Game.initCards(..))) ||
      beforeevent(call(Game.dealCards(player, ..)) && target(player))
  UNTIL
      afterevent(call(*.plantBean(..)))

instance pointcut fieldOneCards<BeanCard> :
      afterevent(call(*.plantBean(1, ..)))
  UNTIL
      beforeevent(call(*.harvest(1)) && call(*.emptyField()) ||
        call(*.harvest(1)) && call(*.remove(bean)) && target(bean)))

instance pointcut fieldTwoCards<BeanCard> :
      afterevent(call(*.plantBean(2, ..)))
  UNTIL
      beforeevent(call(*.harvest(2)) && call(*.emptyField()) ||
        call(*.harvest(2)) && call(*.remove(bean)) && target(bean)))

instance pointcut fieldThreeCards<BeanCard> :
      afterevent(call(*.plantBean(3, ..)))
  UNTIL
      beforeevent(call(*.harvest(3)) && call(*.emptyField()) ||
        call(*.harvest(3)) && call(*.remove(bean)) && target(bean)))

instance pointcut treasuryCards<BeanCard> :
      afterevent(call(*.harvest(..)) && call(*.add(bean)) && target(bean)))
}
```

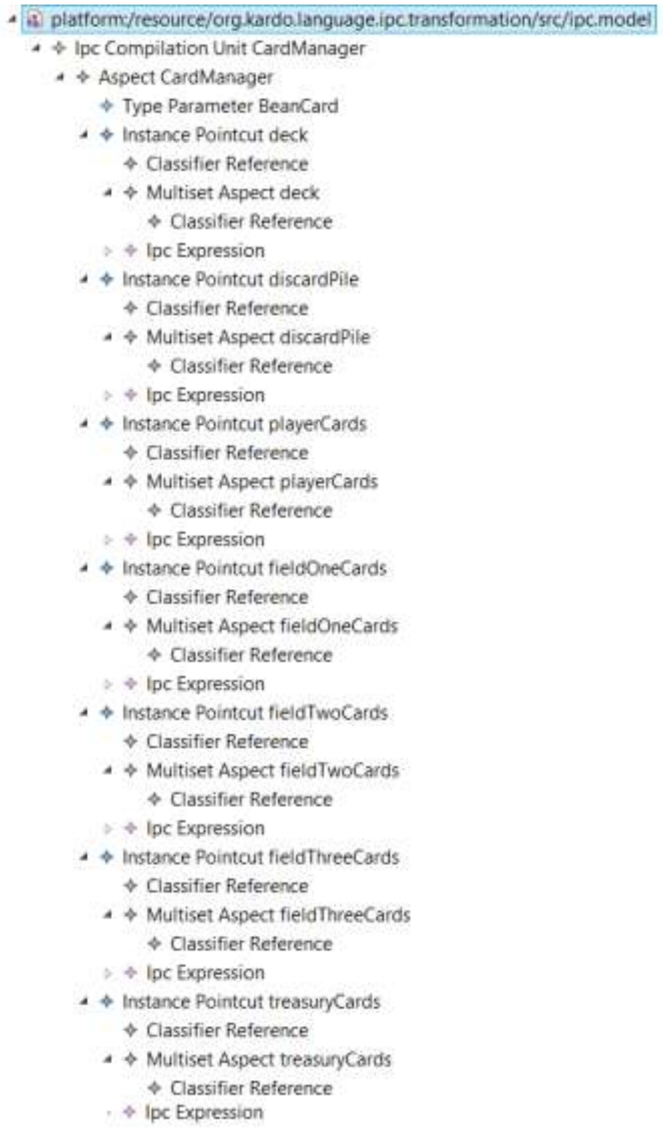
Figure 10. Bohnanza Instance Pointcut Textual Syntax

As figure 10 shows, the textual syntax for Bohnanza is very simple with only a few lines of code required to complete each instance pointcut. Behind the scenes the instance pointcut will then generate a Multiset for each instance pointcut which handles the actual adding and removing cards to and from the set. This also shows the expressiveness of the language to a large extent is dependent on the matching that AspectJ pointcuts can do on objects, as in the end it is a Pointcut Expression that matches the objects to be added or removed.

This textual syntax does immediately show room for improvement if a finer granularity of refined instance pointcuts was defined. For the instance pointcuts regarding fields, the expressions are mostly equivalent apart from the Field they are referencing to; so for the feature complete language, this could be redone if the expressions could be modified slightly.

Composite instance pointcuts were not included at all in the textual syntax because for this application it is simply unnecessary. While some classes do interact with other classes regarding their card lists; hence why some pointcut's add expression is equivalent to another pointcut's remove expression. As one card can only be in one list at a time in this scenario, a composite instance pointcut could be defined instead of one of the existing instance pointcuts, such that if a card is not present in any other set, it should be added to the composite set. However, as this would not affect the overall functionality of the application, nor greatly improve the code quality, I have decided to keep it simple.

The resultant ecore model is as follows:



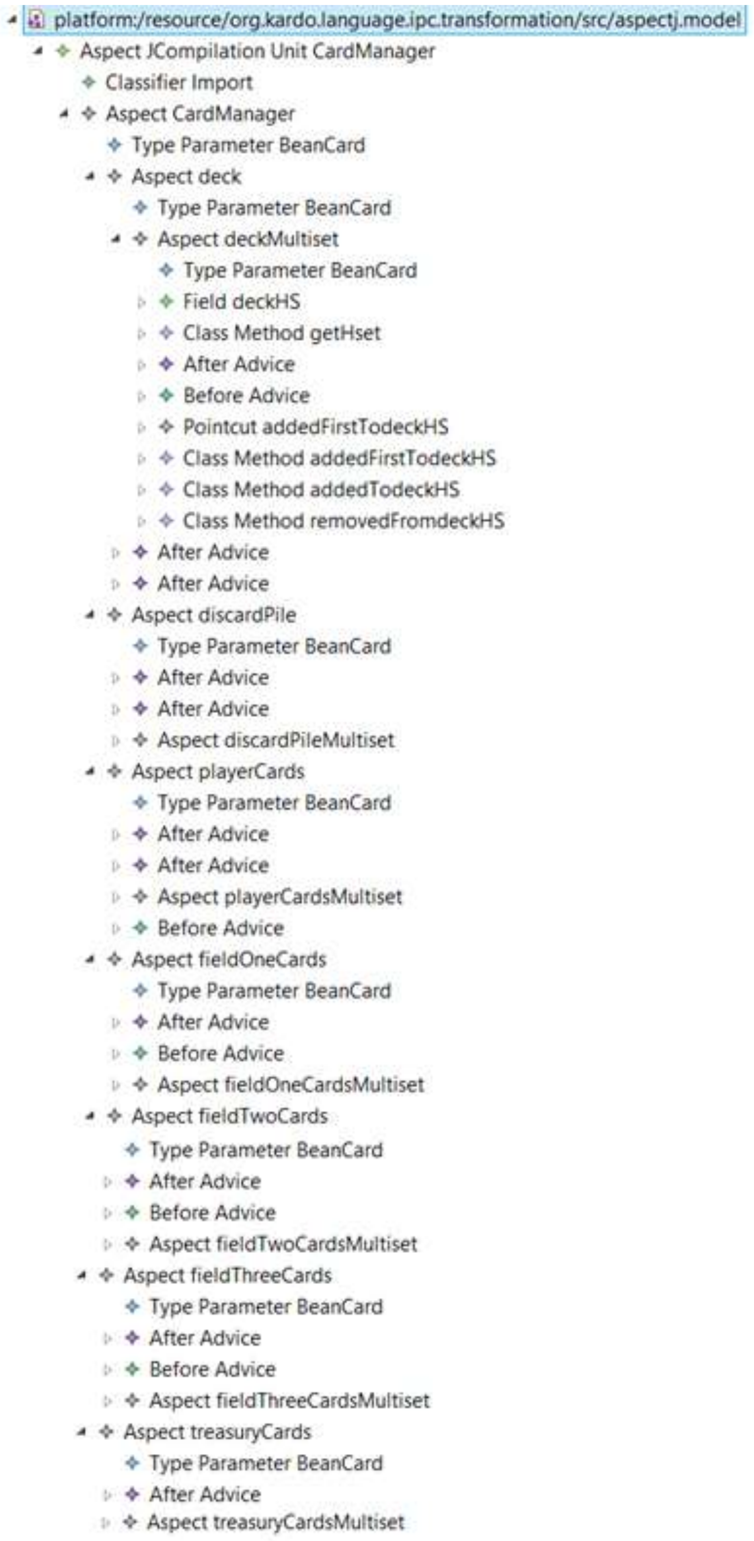
```
platform:/resource/org.kardo.language.ipc.transformation/src/ipc.model
  Ipc Compilation Unit CardManager
    Aspect CardManager
      Type Parameter BeanCard
      Instance Pointcut deck
        Classifier Reference
        Multiset Aspect deck
          Classifier Reference
        Ipc Expression
      Instance Pointcut discardPile
        Classifier Reference
        Multiset Aspect discardPile
          Classifier Reference
        Ipc Expression
      Instance Pointcut playerCards
        Classifier Reference
        Multiset Aspect playerCards
          Classifier Reference
        Ipc Expression
      Instance Pointcut fieldOneCards
        Classifier Reference
        Multiset Aspect fieldOneCards
          Classifier Reference
        Ipc Expression
      Instance Pointcut fieldTwoCards
        Classifier Reference
        Multiset Aspect fieldTwoCards
          Classifier Reference
        Ipc Expression
      Instance Pointcut fieldThreeCards
        Classifier Reference
        Multiset Aspect fieldThreeCards
          Classifier Reference
        Ipc Expression
      Instance Pointcut treasuryCards
        Classifier Reference
        Multiset Aspect treasuryCards
          Classifier Reference
        Ipc Expression
```

Figure 11. Bohnanza Instance Pointcut Model

Figure 11 does show how the textual syntax we produced does convert into a model that can be transformed into AspectJ and then Java. The model also shows that simple Multiset instances are generated automatically for each Instance Pointcut with an identical name and type. While the model above does not show all the properties for each element, nor the details of all the IpcExpression due to the size the resultant model would be to display here; it shows the main entities that are defined when an instance pointcut is made. Also from the Transformation details I discussed previously, this model gives an insight to what AspectJ entities are expected to be produced once it has been transformed.

It is also important to note that under each Ipc Expression there are up to two Sub Expressions, but each are given no identifying name. Similar to with the textual syntax shown above, it is expected that the first Sub Expression is the *ADD* expression, and the optional latter one is the *REMOVE* expression.

While the transformation does need improving still to generate a correct AspectJ model, I can still produce the AspectJ model that would be returned if it did work completely.



```
platform:/resource/org.kardo.language.ipc.transformation/src/aspectj.model
  Aspect JCompilation Unit CardManager
    Classifier Import
    Aspect CardManager
      Type Parameter BeanCard
      Aspect deck
        Type Parameter BeanCard
        Aspect deckMultiset
          Type Parameter BeanCard
          Field deckHS
          Class Method getHset
          After Advice
          Before Advice
          Pointcut addedFirstTodeckHS
          Class Method addedFirstTodeckHS
          Class Method addedTodeckHS
          Class Method removedFromdeckHS
        After Advice
        After Advice
      Aspect discardPile
        Type Parameter BeanCard
        After Advice
        After Advice
        Aspect discardPileMultiset
      Aspect playerCards
        Type Parameter BeanCard
        After Advice
        After Advice
        Aspect playerCardsMultiset
        Before Advice
      Aspect fieldOneCards
        Type Parameter BeanCard
        After Advice
        Before Advice
        Aspect fieldOneCardsMultiset
      Aspect fieldTwoCards
        Type Parameter BeanCard
        After Advice
        Before Advice
        Aspect fieldTwoCardsMultiset
      Aspect fieldThreeCards
        Type Parameter BeanCard
        After Advice
        Before Advice
        Aspect fieldThreeCardsMultiset
      Aspect treasuryCards
        Type Parameter BeanCard
        After Advice
        Aspect treasuryCardsMultiset
```

Figure 12. Bohnanza Transformed AspectJ Model

Figure 12 shows the resulting AspectJ model that would be generated after transformation. While this model I created was not directly generated by my transformation, it still is a clear representation of what is to be expected as the outcome of running the Instance Pointcuts model through the transformation.

The model shows how each Instance Pointcut becomes an Aspect within the main *CardManager* Aspect which did not need to be modified during transformation. The Multisets also become aspects with the slightly modified name to not be identical to the aspect it is nested in. The Multiset is also given all the entities it requires to be able to add and remove objects from the set, as well as return a nonmodifiable set.

One issue the model does display is that each instance pointcut does lose any reference to its Add/Remove expressions, as the events are transformed to Advices, but are put directly within the generated Aspect. This means it becomes unclear whether one of the Advices is used for adding objects, or removing them. This ambiguity is reduced only because instance pointcuts only allow one after, and one before event entity per Add/Remove expression, so if there is two, the first is for add, the latter for remove, and if there is only one advice it must be for adding. But if there are two or more Advices of different types within the Aspect, it currently is not deterministic if they are for adding or removing, so that needs to be improved.

After that the AspectJ model has been created, I can now transform that model into Java using the transformation me and Hristofor have defined. The resultant model is not very important to me for comparing with the AspectJ model, but it is required to be able to compile and run the application. Since there are still some issues and improvements required for the Instance Pointcuts language it is currently not possible to see if the application will run with this implementation, but as no textual syntax nor model errors appeared when creating the implementation I can at least confirm that the implementation is correct in regards to the rules defined in the concrete syntax.

3.4.1.3 Analysis

In this case study, I improved upon the Java implementation by letting Instance Pointcuts manage all the card storage and interactions, rather than have each class individually store a set of cards with accessor methods. In the Deck, Player, Field, and Treasury classes I could now remove all the *ArrayLists* and *LinkedLists* of cards being stored, as well as the accessor methods to them. Taking the Deck as example:

```
private LinkedList<BeanCard> cardDeck;
private ArrayList<BeanCard> discardPile = new ArrayList<BeanCard>();

public ArrayList<BeanCard> getDiscardCards() {
        return discardPile;
}

public LinkedList<BeanCard> getCardDeck() {
        return cardDeck;
}

public void discardCards(ArrayList<BeanCard> beans) {
        if(beans != null && beans.size() > 0) {
                for(BeanCard bean : beans) {
                        discardPile.add(bean);
                }
        }
}

public BeanCard[] getCards(int numberOfCards) {
        if (numberOfCards >= cardDeck.size())
                resetDeck();

        BeanCard[] playerCards = new BeanCard[numberOfCards];
        for (int index = 0; index < numberOfCards; index++) {
                playerCards[index] = cardDeck.poll();
        }
        return playerCards;
}
```

Figure 13. Bohnanza Deck class original Java code

This code can mostly be removed; the last two methods will still exist but can be simplified in regards to accessing the card storage. Any other method from another class referencing one of the lists of cards stored in this class will also need to be updated to get the list of cards from the DeckMultiset Aspect instead.

The removal or modification of such methods will also occur for the other classes for which I made an instance pointcut. While this may require some effort to upgrade an existing application to use my new language; if it initially used the Instance Pointcuts language then that would have been avoided.

Figure 13 does bring up one issue that should be considered as a further improvement to the Instance Pointcut language. That is regarding the discardPile method in Figure 13; it currently takes a list of BeanCards as parameter rather than a single BeanCard. While our instance pointcut and multiset are both of the type 'BeanCard', it currently would not support adding or removing a list of BeanCards from the multiset at once. This is something that would be useful in certain scenarios, when a lot of objects are required to be added or removed in one go, so enabling this functionality should be considered. This could be achieved in several ways, one would be to add or remove each object from the list through use of a loop over the given list. Another method would be to allow complete lists to be stored as one element in the multiset. Or finally, I could make it explicit to a developer that only one object can be added at a time.

The first method would be the most appropriate choice; while Multisets allow for elements in the set to be lists themselves, it would make the structure of the Multiset more complex. Also for removing objects from the set, using a loop and removing each object individually is much safer than removing complete lists at once.

The only issue that would remain when implementing that functionality would be deciding which types of lists or sets I would want to support in the language, as some types of lists are looped through differently (*LinkedLists* and *ArrayLists* for example). So in the end my implementation of the language will stick to the last method and not allow more than one object to be added or removed at a time.

Overall the Instance Pointcuts implementation would offer up to 25% reduction in the lines of code just from the replacement of the Java code to define the list of cards and its accessor methods by instance pointcut definitions. This of course does not include all the code that is generated behind the scenes for the Multiset, which would actually increase the total amount of code written in this implementation by up to 50%. This is more noticeable for small or simple Java lists, since the generated Multiset code is always the same regardless of the size or complexity of the pointcut.

However, the amount of effort necessary to implement the Instance Pointcuts would be lower, since as programmer you would not need to worry about the generated code, and just be able to make use of it. The Instance Pointcuts implementation also offers greater modularity for the application, as all the card storage has been moved to one Aspect rather than be scattered across multiple classes. There will be some dependency of each class to the Multiset Aspect however since the classes will still need to access the list, but that is unavoidable.

3.4.2 Calendar Application

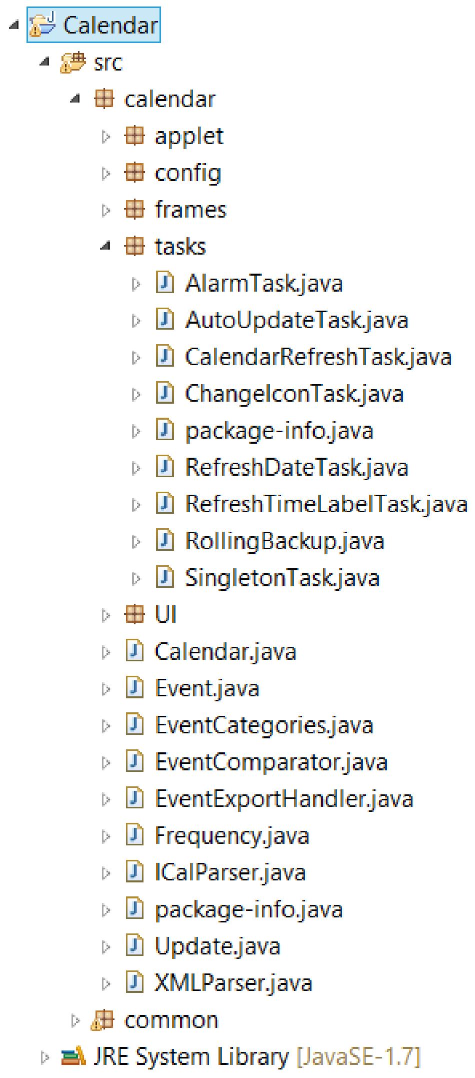
The second case study I decided to improve was a Calendar Application, specifically [26]. Again this application was written in Java, but this time it was not originally written by myself; instead by someone who is likely to be unaware of my Instance Pointcuts language, and would not have implemented the application with that in mind. This calendar was originally implemented in German, but all the source code was written in English which enabled me to understand and use it.

The reason for me choosing a calendar application comes in part from my work experience in developing a modular calendar in a mixture of PHP 5.4 [27] (using the Zend Framework 2 [28] and Apigility [29]), and Javascript [30] (working with a calendar plugin [31]). In developing such a calendar application I observed that calendars can end up storing a lot of appointments and notifications, with the ability for custom filtering or display of different types of appointments, as well as object-level interaction.

Such a calendar would likely have benefits from Instance Pointcuts, since I would be able to improve on the management of appointments within the calendar. This would make it so that accessing or referencing the appointments in a class would be done using the instance pointcuts rather than a different class.

The calendar application I found [26] meets the requirements I was looking for in a calendar, providing sufficient functionality and allowing some level of user-interaction, enough for instance pointcuts to be applicable.

3.4.2.1 The Java Implementation



```
Calendar
  src
    calendar
      applet
      config
      frames
      tasks
        AlarmTask.java
        AutoUpdateTask.java
        CalendarRefreshTask.java
        ChangeIconTask.java
        package-info.java
        RefreshDateTask.java
        RefreshTimeLabelTask.java
        RollingBackup.java
        SingletonTask.java
      UI
      Calendar.java
      Event.java
      EventCategories.java
      EventComparator.java
      EventExportHandler.java
      Frequency.java
      ICalParser.java
      package-info.java
      Update.java
      XMLParser.java
    common
  JRE System Library [JavaSE-1.7]
```

Figure 14. Calendar Java Structure

As Figure 14 shows, the calendar application consists of two main packages, the calendar and the commons packages; the commons package contains classes with reusable logic and user interface entities that are irrelevant to me. The calendar package contains the important class which I will be aiming to improve; the Calendar class. The Calendar class does all the storage of events (which I will call appointments from now on to not confuse with Instance Pointcut Events) and notifications, as well as special types of appointments called *AlarmTasks*.

The other classes handle more calendar-specific logic as well as the parsing of the calendar to XML or ICalendar formats for exporting to other applications. The calendar-specific logic does contain several methods directly related to appointments, which means there is a lot of scattered code among the application.

The sub package 'tasks' within the calendar package will also be affected by my instance pointcuts improvement, since it contains further logic to update or modify appointments within the calendar. The remaining sub packaged however will have little relevance to me.

Overall it does mean I will only be applying instance pointcuts to one class of the application, but due to the amount of dependencies and relations each class has with one another, I should be able to demonstrate what impact even a small number of instance pointcuts can have on a medium-sized application.

3.4.2.2 The Instance Pointcut Implementation

Similar to the Bohnanza case study, the first stage of introducing Instance Pointcuts to the application is to define the textual syntax for the language.

The textual syntax I defined is as follows:

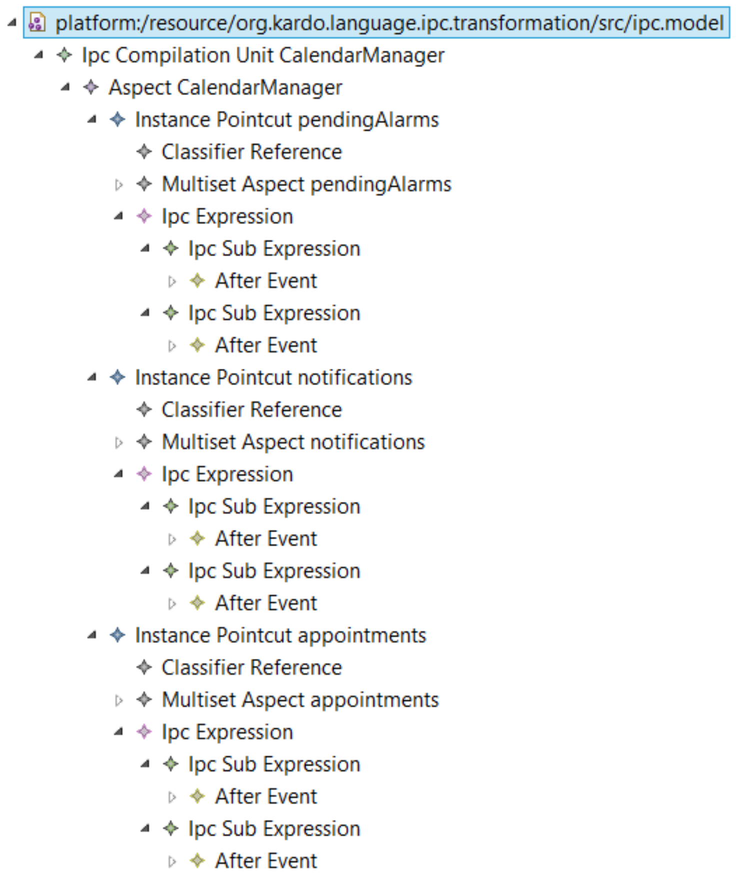
```
package calendar;

aspect CalendarManager {

instance pointcut pendingAlarms<AlarmTask> :
        afterevent(call(Calendar.addAlarmTask(task)) && target(task))
    UNTIL
        afterevent(call(Calendar.removeAlarmTask(task)) && target(task))


instance pointcut notifications<Notification> :
        afterevent(call(Calendar.addCurrentNoti(notification)) && target(notification))
    UNTIL
        afterevent(call(Calendar.removeCurrentNoti(notification)) && target(notification))


instance pointcut appointments<Event> :
        afterevent(call(Calendar.updateFlexibleHolidays(..)) && call(*.add(event)) && target(event) ||
                   call(Calendar.updateStaticHolidays(..)) && call(*.remove(event)) && target(event) ||
                   call(Calendar.addEvent(event, ..)) && target(event))
    UNTIL
        afterevent(call(Calendar.updateFlexibleHolidays(..)) && call(*.remove(event)) && target(event) ||
                   call(Calendar.updateStaticHolidays(..)) && call(*.remove(event)) && target(event))
                   call(Calendar.deleteEvent(event, ..)) && target(event))
}
```

Figure 15. Calendar Instance Pointcuts Textual Syntax

The textual syntax does already show that in this scenario, the add and remove expressions for pendingAlarms and notifications are quite simple, since the java implementation only allows objects to be added or removed to the list in one method. This however does not mean that instance pointcuts will have little impact in this case, since these lists do get accessed just for reading purposes in several instances throughout the application.

The appointments instance pointcut is by far the most complicated instance pointcut, since it not only stores multiple types of appointments, but in this scenario it also makes special cases out of static and flexible holidays. This makes the instance pointcut's expressions quite large, and could potentially be improved by splitting the instance pointcut into smaller ones. A new composite instance pointcut could then be defined as the union of the individual ones, making it overall look clearer. Also if more special case appointment types were defined, it would be easier to add them as a new instance pointcut, and added to the composite instance pointcut in a union. This improvement however would have little overall impact on the usefulness of the language, and only affect the readability, so I have kept the textual syntax as it is.

From the textual syntax I can again define an ecore model for it, which is as follows:



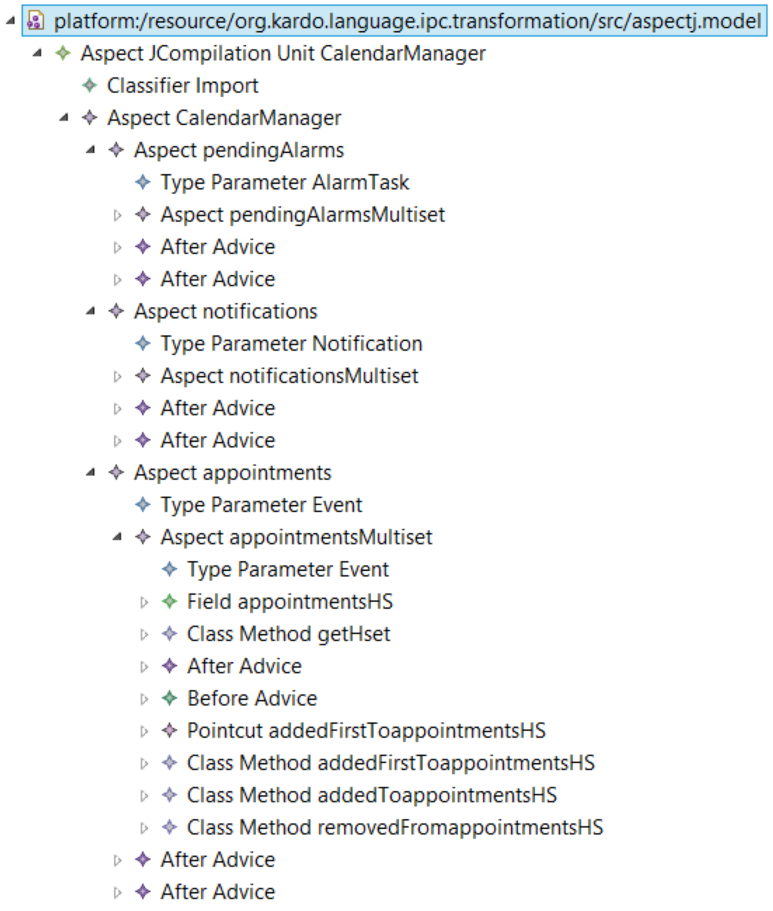
```
platform:/resource/org.kardo.language.ipc.transformation/src/ipc.model
  Ipc Compilation Unit CalendarManager
    Aspect CalendarManager
      Instance Pointcut pendingAlarms
        Classifier Reference
        Multiset Aspect pendingAlarms
        Ipc Expression
          Ipc Sub Expression
            After Event
          Ipc Sub Expression
            After Event
      Instance Pointcut notifications
        Classifier Reference
        Multiset Aspect notifications
        Ipc Expression
          Ipc Sub Expression
            After Event
          Ipc Sub Expression
            After Event
      Instance Pointcut appointments
        Classifier Reference
        Multiset Aspect appointments
        Ipc Expression
          Ipc Sub Expression
            After Event
          Ipc Sub Expression
            After Event
```

Figure 16. Calendar Instance Pointcuts Ecore Model

The model does look very straightforward and very close to the previous case study's model. This is because for each instance pointcut the resultant model entities do end up being very similar, with the only differences being in the name, the Classifier Reference, and the content of the sub expressions. I decided not to display all of the entities for the model again due to the size of the model becoming very large when expanded further than the Event level. Beyond this level in the model would only show the AspectJ Pointcut Expression entities that make up each Event, which is not very relevant to the Instance Pointcuts language.

Also similar to the Bohnanza Instance Pointcut model shown in figure 11, it does make it clear that the Instance Pointcut Sub Expressions lose their reference to belonging to an Add or Remove expression, which will be propagated into the AspectJ model later. This means that the issue should be tackled in this step or the metamodel step of the language, possibly by giving each sub expression a name of either add or remove.

Apart from that, the model does clearly show that not much effort is required to make each instance pointcut entity. And with that model I can then do the transformation into an AspectJ model, and see if the resultant model does stay simple.



```
platform:/resource/org.kardo.language.ipc.transformation/src/aspectj.model
  Aspect JCompilation Unit CalendarManager
    Classifier Import
    Aspect CalendarManager
      Aspect pendingAlarms
        Type Parameter AlarmTask
        Aspect pendingAlarmsMultiset
        After Advice
        After Advice
      Aspect notifications
        Type Parameter Notification
        Aspect notificationsMultiset
        After Advice
        After Advice
      Aspect appointments
        Type Parameter Event
        Aspect appointmentsMultiset
          Type Parameter Event
          Field appointmentsHS
          Class Method getHset
          After Advice
          Before Advice
          Pointcut addedFirstToappointmentsHS
          Class Method addedFirstToappointmentsHS
          Class Method addedToappointmentsHS
          Class Method removedFromappointmentsHS
        After Advice
        After Advice
```

Figure 17. Calendar Transformed AspectJ Model

Figure 17 does show that the resultant model inevitably gets more complicated than the Instance Pointcuts model, mostly because of the Multiset Aspect’s entities becoming defined at the transformation stage. Naturally, all the Multiset Aspect’s entities are necessary for Instance Pointcuts implementation to be successful and have any usefulness to the target application, since within the Aspect it defines the methods for adding and removing objects to the set, as well as having an accessor method for the set.

Furthermore, the nested Aspect structure does make it harder to follow, as there is up to three levels of nesting of Aspects in this model, which decreases readability of the program. It also goes against the design and principle of Java to be a simple programming language [32].

3.4.2.3 Analysis

In this case study, I improved upon the Java implementation by letting Instance Pointcuts manage all of the Calendar's lists regarding AlarmTasks, Notifications, and Appointments. I achieved this by replacing all the list definitions and accessor methods to them with new instance pointcuts, which would then generate the Multiset to store the objects behind the scene. The Calendar class' code that would mostly be removed is the following:

```
private List<AlarmTask> pendingAlarms;
private List<Notification> notis;
private List<Event> events;

public List<Event> getAllEvents() {
        return events;
}

public List<AlarmTask> getAlarmTasks() {
        return pendingAlarms;
}

public List<Notification> getNotifications() {
        return notis;

}

public void removeAlarmTask(AlarmTask x) {
        pendingAlarms.remove(x);
}

public void addAlarmTask(AlarmTask x) {
       pendingAlarms.add(x);
}
```

Figure 18. Calendar class original Java code

This code, along with parts of other methods would all be removed due to becoming redundant in the Instance Pointcuts implementation. Since any code accessing a list defined by an instance pointcut, either to read it or to add or remove an object to the list, has been made redundant due to the functionality the instance pointcut and its multiset aspect bring.

Furthermore, other classes of the Calendar application, specifically ones that modify or update an appointment based on certain logic, will also need to be updated themselves to now use the multiset to access the object. Since these methods want to update the object though, and the multiset will be returning a non-modifiable set, the methods will likely need significant changes to adapt. These methods would first need to have the object be removed from the multiset by matching the Pointcut Expression to remove it defined by the instance pointcut. Once that is done it can update the object, and have the object be added by matching the Pointcut Expression to add it again.

In this scenario, the Instance pointcuts implementation will actually benefit the application more than in Bohnanza, due to the improvement the appointments instance pointcut brings over the Java implementation. Since in the Java application the appointments list gets referenced a number of times by both the Calendar class as well as others; having it be managed by instance pointcuts will significantly reduce the complexity of the Calendar class. On top of having the other lists also being managed by instance pointcuts, the Calendar class would be up to 10% smaller, which given it is over 1,000 lines of code long is significant.

The overall structure of the application will also benefit somewhat of the Instance Pointcuts application, even though instance pointcuts were not directly made for any other class in this scenario. Other classes would now have a lot of dependency on the *CalendarManager* Aspect which contains all the instance pointcuts, but less dependency on the Calendar class itself, making it easier to understand the application.

3.4.3 Evaluation of Case Studies

The case studies have provided very valuable information regarding both the usefulness and potential of the Instance Pointcuts language, as well as the benefits the language provides over Java and AspectJ. In both the case studies I was able to make the coding required to improve the application's functionality using Instance Pointcuts be quite little, as I have explained in their respective analysis chapters.

Looking back at the Research Questions I aimed to answer in chapter 1.4, I can now say that the main question regarding if Instance Pointcuts provide any improvements or advantages to the quality of code compared to the original implementation is answered. I have in both my case studies been able to demonstrate that Instance Pointcuts did make the structure of the application clearer and simpler, and also made an impact on the effort to write the code.

While I set out only to compare an Instance Pointcuts to the original Java application in each case study, I was also able to show due to the intermediate transformation into AspectJ, how the AspectJ implementation for the same application would look like in form of a model. While an AspectJ implementation strategy might have been different for improving the Java application, it still shows the structural benefits Instance Pointcuts provide in managing object sets over AspectJ.

The case studies however also were able to discover any remaining issues or current drawbacks of the Instance Pointcuts language, which I have not only described but also written proposals for to solve the issue and improve the language further. Either way these issues I discovered still did not stop me from being able to demonstrate the usefulness of the language and the benefits the Instance Pointcuts language brings to certain applications.

3.5 Conclusion & Future Work

Even though the Instance Pointcuts language is left with room for improvement in the future; this project has still been successful in regards to being able to answer my Research Questions I set at the beginning. My case studies have shown clear benefits of the language, and how using Instance Pointcuts reduce the amount of coding needed, as well as simplifying the overall structure of the application.

Doing the comparison with the transformed AspectJ model in the case studies has also demonstrated how it would be possible to achieve the same functionality as Instance Pointcuts, but at a vast cost of requiring so many Aspects and visibly a more complicated structure. The Instance Pointcuts language does depend on AspectJ to work, and in the end it comes down to AspectJ's Pointcut Expressions which select the objects to be added or removed from a Multiset, but with Instance Pointcuts it can be done without much effort.

There are of course some tasks and features of the language left to implement to make it complete and further improve it. Due to time constraints however I have left them for future work instead of being able to implement them all.

The first task would naturally be to develop a feature complete implementation of the Instance Pointcuts language. This involves completing the Instance Pointcut Refinement to reach a higher level of granularity – to be able to do refinement on the Expression and Event level of an instance pointcut. In the Bohnanza case study I could already see the potential of the Instance Poincuts language even more if the Field's instance pointcuts could be made to use a refined instance pointcut at the event level of granularity.

Another task to complete the language would be to further improve the transformation of the instance pointcuts to aspectj, this includes proposal to move Add/Remove sub expressions to a nested aspect, and then add some binding of it to the multiset's pointcuts. Furthermore, improve the composite instance poincut to make each Ipc term be placed with a nested aspect for Union/Intersection, then add postprocessing logic, or transformation logic to decide how the composite multiset should be populated from that information.

Also, as brought up in the evaluation of the case studies, it would be a nice extra feature of the language if it could support adding or removing more than one object at a time to or from a multiset. While some decisions would have to be made on which of the proposed methods to follow for implementing the feature; it would become quite useful for certain applications where multiple objects are often required to be added or removed in one go.

With a feature-complete language, do an analysis and comparison of an implementation of an application in Instance Pointcuts to AspectJ. While I currently already demonstrate how the current Instance Pointcuts language can transform an Instance Pointcuts application to AspectJ; we specifically want to demonstrate how a new feature of set monitoring. This is where the language implicitly defines two set change pointcuts for each multiset when an instance is added or removed from the application. A comparison can then be made how Instance Pointcuts improves on the set monitoring functionality in an application compared to an AspectJ implementation. Furthermore, this would enable the ability to attach advices to events related to instance pointcuts, so that an advice can make use of the pointcuts that are implicitly defined in the multiset. Advices would allow adaptation to occur to an object that is about to be added or removed from an existing multiset, and allow it to happen before or after the object is added/removed.

As a bonus, some extra tasks that would further improve the language would be to do an analysis and comparison of the performance of the language after being compiled and executed from the instance pointcut implementation, and the original implementation. Also, doing more thorough testing of the Instance Pointcuts metamodel and transformation would help to prove that the language is sound

Finally, as Kardelen investigated in her thesis [24], and I discussed in my previous study [20], some consideration should be done in regards to the benefits of combining the language with an alternative language such as typestates to grant more functionality.

4. References

[1] The AspectJ Team, “The AspectJTM 5 Development Kit Developer’s Notebook.” [Online]. Available: http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html. [Accessed: 14-Sep-2014].

[2] J. Bézivin, “On the unification power of models,” Softw. Syst. Model., vol. 4, no. 2, pp. 171–188, 2005.

[3] R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap,” in 2007 Future of Software Engineering, Washington, DC, USA, 2007, pp. 37–54.

[4] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: eclipse modeling framework. Pearson Education, 2008.

[5] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, “Derivation and refinement of textual syntax for models,” in Model Driven Architecture-Foundations and Applications, 2009, pp. 114–129.

[6] DevBoost, “EMFText User Guide.” [Online]. Available: https://github.com/DevBoost/EMFText/blob/master/Core/Doc/org.emftext.doc/pdf/EMFTextGuide.pdf?raw=true. [Accessed: 15-Sep-2014].

[7] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, Jamopp: The java model parser and printer. Techn. Univ., Fakultät Informatik, 2009.

[8] C. Allan, P. Avgustinov, A. S. Christensen, B. Dufour, C. Goard, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, and C. Verbrugge, “Abc the aspectBench Compiler for aspectJ a Workbench for Aspect-oriented Programming Language and Compilers Research,” in Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 2005, pp. 88–89.

[9] N. Nystrom, M. R. Clarkson, and A. C. Myers, “Polyglot: An extensible compiler framework for Java,” in Compiler Construction, 2003, pp. 138–152.

[10] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, “Optimizing Java bytecode using the Soot framework: Is it feasible?,” in Compiler Construction, 2000, pp. 18–34.

[11] R. Toledo and É. Tanter, “A Lightweight and Extensible AspectJ Implementation.,” J UCS, vol. 14, no. 21, pp. 3517–3533, 2008.

[12] L. Hendren, O. De Moor, A. S. Christensen, and others, “The abc scanner and parser, including an LALR (1) grammar for AspectJ,” 2004.

[13] Eclipse Foundation, “Eclipse Modeling - MMT.” [Online]. Available: http://www.eclipse.org/mmt/. [Accessed: 14-Sep-2014].

[14] D. S. Kolovos, R. F. Paige, and F. A. Polack, “The epsilon transformation language,” in Theory and practice of model transformations, Springer, 2008, pp. 46–60.

[15] I. Kurtev, “State of the art of QVT: A model transformation language standard,” in Applications of graph transformations with industrial relevance, Springer, 2008, pp. 377–393.

[16] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” Sci. Comput. Program, vol. 72, no. 1, pp. 31–39, 2008.

[17] “Using black box implementations in Eclipse QVTo | F. Levy Siqueira.” [Online]. Available: http://www.levysiqueira.com.br/2012/02/black-box-eclipse-qvto/. [Accessed: 15-Sep-2014].

[18] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, “The Epsilon Book.” [Online]. Available: http://www.eclipse.org/epsilon/doc/book/. [Accessed: 14-Sep-2014].

[19] The AspectJ Team, “AspectJ 5 Quick Reference.” [Online]. Available: http://www.eclipse.org/aspectj/doc/released/quick5.pdf. [Accessed: 15-Sep-2014].

[20] G. de Heer, “Research Topics: Instance Pointcuts.”

[21] K. Sullivan, H. Rajan, “Eos: Instance-Level Aspects for Integrated System Design.” [Online]. Available: http://delivery.acm.org/10.1145/950000/940111/p297-rajan.pdf?ip=130.89.169.166&id=940111&acc=ACTIVE%20SERVICE&key=0C390721DC3021FF.7DEDEACE9AC2380A.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=421786143&CFTOKEN=38472676&__acm__=1394897712_278a2afd14b0867dcc01d2d28e2539d2. [Accessed: 15-Nov-2014].

[22] S. Cluet, “Designing OQL: Allowing Objects to be Queried.” [Online]. Available: http://ac.els-cdn.com/S0306437998000131/1-s2.0-S0306437998000131-main.pdf?_tid=c7a83f36-ac56-11e3-8d9c-00000aacb35d&acdnat=1394897626_88536082b7a470d030a5005582df73dc. [Accessed: 15-Nov-2014].

[23] R. DeLine, M. Fähndrich, “Typestates for Objects.” [Online]. Available: http://download.springer.com/static/pdf/3/chp%253A10.1007%252F978-3-540-24851-4_21.pdf?auth66=1395070288_f577be2c570465e003bc54c7ad932866&ext=.pdf. [Accessed: 15-Nov-2014].

[24] K. Hatun, “Non-intrusive instance level software composition.” [Online]. Available: http://doc.utwente.nl/89423/1/thesis_K_Hatun.pdf. [Accessed: 15-Nov-2014].

[25] G. de Heer, R. Balk, “Patterns of Software Development: Bohnanza Report.”

[26] Jsteltze, “Java-Calendar.” [Online]. Available: https://github.com/devypt/Kalender. [Accessed: 15-Nov-2014].

[27] The PHP Team, “PHP Documentation.” [Online]. Available: http://php.net/docs. [Accessed: 15-Nov-2014].

[28] The Zend Framework Team, “Programmer’s Reference Guide of Zend Framework 2.” [Online]. Available: http://framework.zend.com/manual/2.0/en/index.html. [Accessed: 15-Nov-2014].

[29] The Zend Framework Team, “Apigility Documentation.” [Online]. Available: https://apigility.org/documentation. [Accessed: 15-Nov-2014].

[30] The Javascript Team, “Javascript Reference.” [Online]. Available: http://www.w3schools.com/jsref/. [Accessed: 15-Nov-2014].

[31] A. Shaw, “Full Calendar Plugin.” [Online]. Available: http://fullcalendar.io/docs/. [Accessed: 15-Nov-2014].

[32] R. Nageswara Rao, “Core Java: An Integrated Approach: Covers Concepts, Programs, and Interview Questions.” Kogent Solutions, 2008, p 178.