ESCUELA POLITÉCNICA SUPERIOR DE MONDRAGON UNIBERTSITATEA
MONDRAGON UNIBERTSITATEKO GOI ESKOLA POLITEKNIKOA
MONDRAGON UNIVERSITY FACULTY OF ENGINEERING

Trabajo fin de grado presentado para la obtención del título de
Titulua eskuratzeko gradu bukaerako lana
Final degree project for taking the degree of

GRADO EN INGENIERÍA INFORMÁTICA
INFORMATIKAKO INGENIARITZA GRADUA
DEGREE IN COMPUTER ENGINEERING

Título del Trabajo *Lanaren izenburua* Project Topic

REALIZING FEATURE REFACTORING IN SOFTWARE PRODUCT LINES

Autor *Egilea* Author UGAITZ AMOZARRAIN PEREZ

Curso *Ikasturtea* Year 2011/2012

Título del Trabajo *Lanaren izenburua* Project Topic

REALIZING FEATURE REFACTORING IN SOFTWARE PRODUCT LINES

Nombre y apellidos del autor
Egilearen izen-abizenak
Author's name and surnames
AMOZARRAIN PEREZ, UGAITZ

Nombre y apellidos del/los director/es del trabajo
Zuzendariaren/zuzendarien izen-abizenak
Project director's name and surnames
ROBERTO E. LOPEZ-HERREJON
ETXEBERRIA, LEIRE

Lugar donde se realiza el trabajo
Lana egin deneko lekua
Company where the project is being developed
JOHANNES KEPLER UNIVERSITY LINZ

Curso académico
Ikasturtea
Academic year
2011/2012

Documentación entregada en Secretaría Académica
Idazkaritza Akademikoan entregatutako dokumentazioa
Documentation handled to Academic Secretary

Memoria *Txostena* Memory

CD-Rom

A rellenar por MGEP *MGEP-ek betetzekoa* To be fullfilled by MGEP

El autor/la autora del Trabajo Fin de Grado, autoriza a la Escuela Politécnica Superior de Mondragon Unibertsitatea, con carácter gratuito y con fines exclusivamente de investigación y docencia, los derechos de reproducción y comunicación pública de este documento siempre que: se cite el autor/la autora original, el uso que se haga de la obra no sea comercial y no se cree una obra derivada a partir del original.

Gradu Bukaerako Lanaren egileak, baimena ematen dio Mondragon Unibertsitateko Goi Eskola Politeknikoari Gradu Bukaerako Lanari jendeaurrean zabalkundea emateko eta erreproduzitzeko; soilik ikerketan eta hezkuntzan erabiltzeko eta doakoa izateko baldintzarekin. Baimendutako erabilera honetan, egilea nor den azaldu beharko da beti, eragotzita egongo da erabilera komertziala baita lan originaletatik lan berriak eratortzea ere.

Abstract

Organizations must create product lines rather than single products more often than not. Tools and techniques for software development focus on single products. This has created a need for new techniques for *Software Product Line* development. One of such techniques *Feature Oriented Software Development* requires that the program is divided into modules, called *features*.

This work provides seven transformation rules written in the TXL programming language that allow for *feature extraction* from previously annotated code. These transformation rules were created from previously defined patterns found while manually refactoring code. A Java program is also provided to handle the calls to the TXL interpreter. The feature extraction leaves the code of each feature in a separate folder and then FeatureHouse will have to be used to create different variants.

To show that the extraction rules work as intended, and that there are no problems when composing a program from extracted code, we tested two case studies. The first was VODPlayer with 10 features and ∼4.5KLoC. The second one was ArgoUML with 25 features and ∼120KLoC.

From our work we suggest that the extraction rules should be integrated in an IDE. This way the process of annotating the code would be easier.

Contents

1 Introduction **1**
- 1.1 Background and state-of-the-art . . . 2
- 1.2 Goals . . . 2
- 1.3 Phases . . . 3
- 1.4 Chapter description . . . 3

2 Software Product Lines **4**
- 2.1 Feature-Oriented Domain Analysis . . . 7
- 2.2 Feature Oriented Software Development . . . 8
- 2.3 Using FOSD to compose programs . . . 8
  - 2.3.1 Templates . . . 9
  - 2.3.2 AHEAD tool suite (ATS) . . . 10
  - 2.3.3 FeatureHouse . . . 11
- 2.4 Aspect Oriented Programming (AOP) . . . 13
  - 2.4.1 AspectJ . . . 13

3 TXL (Turing eXtender Language) **15**
- 3.1 Grammar definition . . . 16
  - 3.1.1 Tokens . . . 17
  - 3.1.2 Compounds . . . 18
  - 3.1.3 Keys . . . 18
  - 3.1.4 Comments . . . 18
  - 3.1.5 Nonterminal definition . . . 19
- 3.2 Transformation rules . . . 20
  - 3.2.1 Parameters . . . 21
  - 3.2.2 Global variables . . . 21
  - 3.2.3 Constructor and deconstructors . . . 21
  - 3.2.4 Conditions . . . 22
  - 3.2.5 Conditional rules . . . 23

4 Patterns & Implementation **24**
- 4.1 Rule application order . . . 26
- 4.2 Transformation rules . . . 27
  - 4.2.1 Extract with hook . . . 28
  - 4.2.2 Extract blocks . . . 31
  - 4.2.3 Remove different feature class body declarations . . . 32
  - 4.2.4 Remove non marked class body declarations . . . 33

4.2.5 Remove modifiers . . . 34
4.2.6 Remove imports . . . 35
4.2.7 Erase if empty . . . 36
4.3 Result . . . 36
4.4 Statistics . . . 38

5 Case studies 39
5.1 VOD Player Running Example . . . 40
5.1.1 Refactoring code criteria . . . 41
5.1.2 Identifying features in code . . . 41
5.2 ArgoUML . . . 42
5.3 Statistics . . . 43

6 Conclusions and future work 45
6.1 Conclusions . . . 46
6.1.1 Implementation . . . 46
6.1.2 Annotation . . . 46
6.1.3 Viability of TXL . . . 46
6.2 Future work . . . 46

I TXL source code 48
I.a Main file . . . 49
I.b Extract block with hook . . . 50
I.c Extract block . . . 52
I.d Remove non featured class body declarations . . . 55
I.e Remove different feature class body declarations . . . 56
I.f Remove modifiers . . . 57
I.g Remove imports . . . 59
I.h Erase if empty . . . 59
I.i Grammar overrides . . . 61

Bibliography 63

List of Figures

2.1 Dell Latitude©E5520 5
2.2 Basic SPL concepts 6
2.3 Template engine 10
2.4 Example of module and aspect 13

3.1 TXL process 16

4.1 Program Flow 26
4.2 Statement removal example 26

5.1 VODPlayer feature model 41
5.2 ArgoUML feature model as seen in [16] 43
5.3 Annotations statistics comparison 44

List of Tables

4.1 Relation between extraction patterns and TXL rules 25
4.2 TXL statistics 38

5.1 VOD Player Requirements, Features, and Refactoring Criteria 40
5.2 ArgoUML Requirements, Features, and Refactoring Criteria 42
5.3 Annotation statistics 44
5.4 Execution time statistics 44

Chapter 1

Introduction

Contents

1.1 Background and state-of-the-art 2
1.2 Goals 2
1.3 Phases 3
1.4 Chapter description 3

This section gives an introduction to the work done during my stay at Johannes Kepler Universität. First, the background and the state of the art is given for the thesis, explaining the benefits of using software product lines on software development. After this, we describe the goals that we want to achieve with this work and we list the phases we followed to achieve those goals. Finally, a general description of the thesis is given, describing briefly what is written in each section.

1.1 Background and state-of-the-art

Software development is the process of writing and maintaining source code, but it can also be used to reference everything that is done from the conception of the software to the to the final manifestation of the software. Therefore, software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products. The way to apply these different processes has changed in the last years as different tools for software development have been created, to support the development of bigger and more complex programs, which have to run on heterogeneous systems and meeting different requirements. All this has changed the way software is developed.

One of these changes is the creation of *Software Product Lines (SPL)*. When different product can be assembled using common *features* (increments on program functionality) [23,24] a SPL is created, usually this happens when having a core application and different variants of the program are needed. The creation of a SPL improves the quality of the software and decreases the costs of creating products. However, the problem that people face when creating a SPL is that the program is already written and the features are mixed together. To the process of obtaining the different features we call *feature refactoring*. Once the we have each feature correctly defined we can create new ones and integrate it on the core program. We have different and well tested tools to compose a program with different features, but there is not a tool that allows us to get the different features from an already composed program in a feasible way. We have already the definition of different patterns found when doing the feature refactoring and now we have to find a way to realize those patterns in source code.

1.2 Goals

1. Analyse the viability of the TXL programming language to implements the refactoring patterns defined in a previous work.

2. Realize the patterns. We provide 7 transformations and a way to annotate code that allow for the application of the refactoring patterns.

3. Provide a way to apply the refactorings in large projects. We do this by providing a Java program that handles the files and the calls to TXL.

4. Validate that the created transformation rules are able to refactor code into different features and it is possible to compose the original program with all its variants with the created features.

1.3 Phases

1. Project management.

   Weekly meeting took place in order to assure that the deadlines are met and the project work is going forward. The meeting were held on Fridays nearly every week.

2. Initial research of the state-of-the-art.

   An initial research was done to learn the state-of-the-art of software product lines and program composition tools.

3. TXL transformations implementation.

   The refactoring patterns had to be implemented using the TXL programming language. TXL was also evaluated to see if it was viable to apply the refactorings using it.

4. Test the created transformation rules.

   We tested the transformation rules written in TXL with two different programs to see if it is possible to use the feature extraction rules in normal projects.

5. Thesis writing and presentation.

   The thesis was written through the year though most of it has been written at the end. The presentation will take place in Mondragon Unibertsitatea in July 17th, 2012.

1.4 Chapter description

Chapter 1 is this introduction.

Chapter 2, *Software Product Lines*, gives a background on SPL and explains different tools for software composition inside feature oriented software development.

Chapter 3, *TXL - Turing eXtender Language*, gives a general idea of the TXL programming language and how its transformations work.

Chapter 4, *Patterns & Implementation*, explains the patterns that were implemented, how each of them is annotated on the source code of the program and how they work.

Chapter 5, *Case studies*, describes the programs that were used to test the transformations and the results obtained.

Chapter 6, *Conclusions and future work*, analyses the work done and suggests further development lines.

Chapter 2

Software Product Lines

Contents

2.1 Feature-Oriented Domain Analysis 7

2.2 Feature Oriented Software Development 8

2.3 Using FOSD to compose programs 8

2.3.1 Templates 9

2.3.2 AHEAD tool suite (ATS) 10

2.3.3 FeatureHouse 11

2.4 Aspect Oriented Programming (AOP) 13

2.4.1 AspectJ 13

A product line is a set of related products developed from a shared set of assets [13]. These product can have shared characteristics if the same asset is used in different product or they can have unique characteristics if one asset is used only for one product. Using product lines to create different products improves reuse as same assets are used to create different products, supports customization, allowing the customers to choose the product they want, and reduces the time it takes to market a product since the assets are already created.

As an example of a product line we can use a computer. To build a computer we need a fixed set of assets, consisting of a power supply, motherboard, processor, RAM memory, graphics card, a storage unit and a case, we have also some optional parts as a sound card, wifi card, etc. This allows us to build very configurable computers since we can choose from a huge set of assets. Two computers can look the same, and be of the same series, but be different models and have completely different characteristics and prices. For example dell offers customizable laptops on their website some of them go from 499$ to over 1500$ depending on what screen, microprocessor, battery, etc. the user chooses, and all of them use the same case.

Figure 2.1: Dell Latitude©E5520
Taken from Dell's website www.dell.com

If we want to apply this to software development we get Software Product Lines (SPL). A SPL is a set of similar programs which are differentiated by the features each of them implement. When a company starts to create a product the problem of how the product is going to evolve arises. This happens on all kind of companies. The best option for a company is to have different variations of the same product. The ability to create all these variations is called mass customization and is usually a computer-aided process. The customization on software is based on the reutilization of code. The difference between the software reuse on a SPL and the reuse that was done before is that when designing a SPL, the code has to be oriented towards reuse from the beginning, predicting in witch

cases the reusable software has to be integrated on the core, instead of creating libraries and looking for similar features on older code.

A software product line can be described using four simple concepts. These concepts [5] illustrate the key objectives when building a software product line, to capitalize on commonality, avoiding duplication and divergence, and manage variation by defining the available options. These objectives allow to reduce time, effort, cost and complexity when creating and maintaining a line of similar products.

- *Software asset inputs*
  This concept includes the requirements, source code components, test cases, architecture, and documentation, that are the software assets that can be configured and composed in different ways to create all the different products of a product line. Each of the assets has a well defined role on the product line. To be able to offer variation on the products some assets may be optional and some others may have variation points that can be used to provide different behaviour.

- *Decision model and product decisions*
  A decision model describes all the optional and variable features that can be used to create products in the product line. Each product in the product line is defined by its product decisions, the choices of the optional and variable features in the decision model.

- *Production mechanism and process*
  The means for composing and configuring products from the inputs. The previously made decisions are used to determine which assets to use as inputs and how they are configured.

- *Software product outputs*
  These are the collection of all products that can be produced using a product line. The scope of the product line is determined by the set of software outputs that can be made using the assets and decision model.

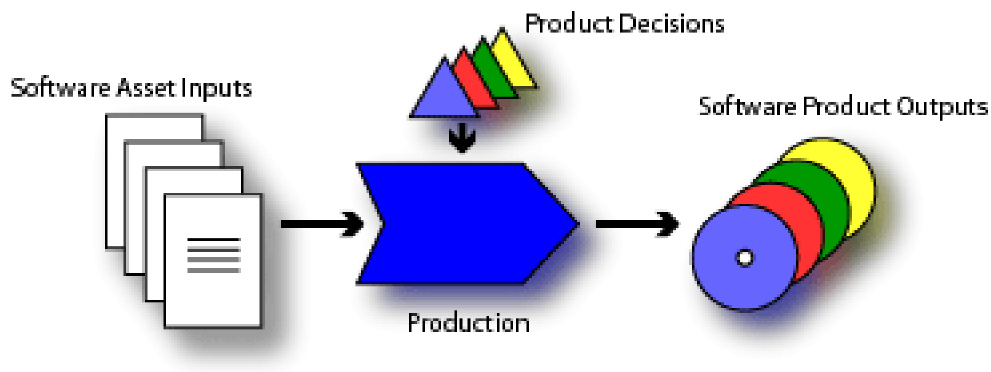

Figure 2.2: Basic SPL concepts

There are two parallel processes in the development of a Software Product Line.

Domain engineering: Its the process of reusing acquired knowledge in the production of new software systems, this knowledge is acquired and reused inside a family of similar systems. Significant savings may be achieved by reusing portions of previous systems when building new variants. Domain engineering has 3 phases: domain analysis, domain design and domain implementation. The first one, is used to define the domain and to identify common and varying points in the domain, the common points are the ones that can later be reused through the whole domain. In the second one, domain design, the domain model that is created in the analysis is used to produce a generic architecture in which all the systems in the domain are included. The last phase, domain implementation, is about creating tools to be able to generate customized programs in the domain.

Application engineering: Its the process of using the results from domain engineering to create different products of the same software product line. It has three phases which are fed by their corresponding phases in domain engineering: analysis, design and implementation.

2.1 Feature-Oriented Domain Analysis

Feature Oriented Domain Analysis(FODA) is a method for domain analysis that uses feature modelling. The feature model is a representation of all the products that are in a software product line. A feature is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [19]. A feature model has four main elements:

- *Feature diagram:* Its the most important part of the model. Its a tree like structure with a root defining the concept that is going to be built, different nodes for each option or feature, and edges that represent the relationships between nodes.
- *Feature definition:* Each node has to have a description of what it does.
- *Composition rules for features:* Rules that define valid combinations of features.
- *Rationale for features:* Indicates the reasons why a feature has been chosen over the others.

There are four kinds of relations between features:

- *Mandatory:* A feature that is indicated as mandatory will be included if the parent is also included. It is denoted whit a filled circle on the child side of the relation.
- *Optional:* It describes a feature that can be included if the parent is also included. It is denoted with an empty circle on the child side of the relation.
- *Or:* Different nodes can be related as optional, but at least one of them must be chosen. It is denoted with a filled arc between all the edges that are included on the relation on the parent side.

- *Alternative(xor):* Only one of the features must be selected. It is denoted with an empty arc between all the edges that are included on the relation on the parent side.

To further describe relations between features FODA also uses two types of composition rules:

- *Require rules:* Describe interactions between features, defining when a feature need another one.
- *Mutually exclusive rules:* Describe when two or more features can not be together on a product.

2.2 Feature Oriented Software Development

Feature Oriented Software Development (FOSD) is a paradigm that allows the creation of customizable and variable code [11, 12, 27]. FOSD arose from layer based designs and levels of abstraction that were used in the late 1980s. It focuses on features and describing how different combinations of features compose a program.

In FOSD a program is defined a stack of layer where each layer adds functionality to the previously composed layers. Different compositions of these layers create different programs. There was a need for a language to express the compositions and elementary algebra was chosen.

There are different tools that allow the composition of programs using FOSD, but all of them share the idea of using algebra to define program composition. Since each layer was a function that added new code to the existing program, the programs design can be defined using an expression.

2.3 Using FOSD to compose programs

One of the first tools to model programs using FOSD was GenVoca, later this was extended in AHEAD. In GenVoca a model of a software product line is an algebra that offers a set of operations, where each operation implements a feature. GenVoca contains different elements:

- Base programs or transformations called values:

```
1 f    -- base program with feature f
2 h    -- base program with feature h
```

- Functions that represent program refinements:

```
1 i • x    -- adds feature i to program x
2 j • x    -- adds feature j to program x
3 The symbol • denotes function application
```

- An application with multiple features is defined with an equation:

```
1  p1 = j • f          – program p1 has features j and f
2  p2 = j • h          – program p2 has features j and h
3  p3 = i • j • h      – program p3 has features i, j, and h
```

Constraints also exist when defining equations, this means that not all possible equations or all possible feature combinations will be valid.

The other tool, AHEAD [14], which stands for *Algebraic Hierarchical Equations for Application Design* is an extension to GenVoca. This extension is done in two ways, first the internal structure of GenVoca is defined as tuples. Every program has multiple representations, such as source, documentation, bytecode, and makefiles. For example if we have a program g with source s_g, documentation d_g and makefile m_g the program p is modelled by the tuple $f = [s_g, d_g, m_g]$.

Second, AHEAD expresses features as nested tuples called deltas, deltas can be program refinements, extensions or interactions. As an example we have a feature f that has some extensions to source code by $\triangle s_f$ and extends the documentation by $\triangle d_f$. The tuple of deltas for feature f is modelled by $f = [\triangle s_f, \triangle d_f]$, this is called a delta tuple. Each element of a delta tuple can be also delta tuples themselves, for example, refinement $\triangle s_f$ defines changes made to each source file of the program. As for program composition:

```
1  p = f • g
2    = f = [△s_f, △d_f] • f = [s_g, d_g, m_g]
3    = f = [△s_f • s_g, △d_f • d_g, m_g]
```

In this example the source of program p is composed by the extension that feature f does to program g ($f = [\triangle s_f \bullet s_g$) and the documentation also changes by the extension of feature f ($\triangle d_f \bullet d_g$), but the makefiles are not modified. As the elements of a delta tuple can also be delta tuples, the program composition is recursive.

There are several tools that use AHEAD metodology for program composition some of them are the AHEAD tool suite [11], FeatureHouse [9], and CIDE [21].

2.3.1 Templates

Templates are not composition tools oriented to FOSD, but they can be compared to see what advantages FOSD has over templates.

Templates are text files that contain commands and code fragments. These text files are instantiated by a template processor and code particular to a program variant is obtained. An example of a template engine is the Velocity template engine [8]. Figure 2.3 shows how the template engine merges both the data and the template code to obtain the desired variant.

Templates can be used for different purposes, that can vary from pretty printing to code generation from XML files, the purpose depends on the engine that is used. However since template engines are very flexible they often enable unconventional uses.

As the templates can be easily changed and the output code generation is simple they are used in fast development cycles. Most template engines are interpreted and use imperative scripting languages in the template definition. The main problem with templates is that their code is hard to understand and it is very difficult to imagine how the output code will look after processing it with a template engine.

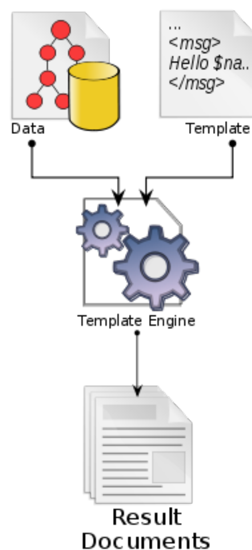

Figure 2.3: Template engine

2.3.2 AHEAD tool suite (ATS)

It is a tool for feature oriented programming that uses step-wise refinements [14]. The concept of step-wise refinements is simple, complex programs are composed from simple programs by incrementally adding details. This is also the concept behind ATS. The main characteristics of AHEAD are:

- Features are the primary units of a program.
- Features encapsulate different representations of a program (code, makefiles, documentation, tests, etc.).
- Each feature contains code corresponding to different classes.
- Features are operators and they are used to create an application.
- Features are composed by merging corresponding program representations.

The AHEAD tool suite is a set of tools, written in Java, that allow for development using AHEAD, for programs written in Java. It uses a language which is an extension of Java called Jak, short for Jakarta. A program using ATS is separated in different folders, one for each feature. Each feature folder will have *.jak* files that define new classes or refine previously defined ones, adding new functionality. When composing a program a equation has to be provided to the tool, which defines the program composition, and the tool will use its composer to create the *.java* files from the composition of the selected features.

Listing 2.1 shows how two different features are merged together. A `refines` statement is used on line 13 to tell the composer that class `C` is going to be refined, and the `Super().m1()` statement on line 16 references the function previously defined on feature `A`.

```
1  //Feature A
2  layer A;
3  public class C {
4    int i;
5    double d;
6    public void m1(){
7      stmt1;
8      stmt2;
9    }
10 }
11 //Feature B
12 layer B;
13 refines class C {
14   float f;
15   public void m1(){
16     Super().m1();
17     stmt3;
18   }
19   public void m2(){
20     stmt4;
21   }
22 }
23 //After composition
24 public class C{
25   int i;          //feature A
26   double d;        //feature A
27   float f;         //feature B
28   public final void m1$$A() {
29     stmt1;
30     stmt2;
31   }
32   public void m1(){
33     m1$$A();       //feature A
34     stmt3;         //feature B
35   }
36   public void m2(){   //feature B
37     stmt4;
38   }
39 }
```

Listing 2.1: ATS composition example

2.3.3 FeatureHouse

FeatureHouse [9] provides a general architecture composed by a framework and a tool-chain for language-independent composition. It is the successor of AHEAD tool suite

[11]. Unlike ATS, FeatureHouse has a language-independent model so extensions can be made to the tool to allow for new programming languages. For the program composition FeatureHouse uses a an improved version of a tool called FSTComposer. This composer relies on a structure of software artifacts, called the *feature structure tree* (FST), that represent the modular structure of a programming language. FeatureHouse is written in Java.

Listing 2.2 shows an example of how composition works using FeatureHouse. The program is composed of two features that make changes on one class `C`, feature `A` is on lines 2-9, with two fields and a method, and feature `B` is between lines 11-20. Feature `B` adds a field and a method and refines method `m1` from feature `A` using the `original` keyword. In the result, lines 22-34, the composed class `C` has the three fields from both features, on method `m1` substitutes the `original` keyword by the contents of method `m1` on feature `A`. Finally, method `m2` from feature `B` is added to the composed class. For further detail see [9].

```
1  //Feature A
2  public class C {
3    int i;
4    double d;
5    public void m1(){
6      stmt1;
7      stmt2;
8    }
9  }
10 //Feature B
11 public class C {
12   float f;
13   public void m1(){
14     original();
15     stmt3;
16   }
17   public void m2(){
18     stmt4;
19   }
20 }
21 //After composition
22 public class C{
23   int i;          //feature A
24   double d;       //feature A
25   float f;        //feature B
26   public void m1(){
27     stmt1;        //feature A
28     stmt2;        //feature A
29     stmt3;        //feature B
30   }
31   public void m2(){  //feature B
32     stmt4;
```

```
33   }
34 }
```

Listing 2.2: FEATUREHOUSE composition example

2.4 Aspect Oriented Programming (AOP)

Thought *Aspect Oriented Programming* and *Feature oriented programming* are two different paradigms AOP was analysed in order to see if it has nay advantages over FOP.

Aspect oriented programming is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns from the program [17, 20]. A *cross-cutting concern* is a part of a program that affects other concern, an example of this is the logging of a program. AOP forms a basis for aspect oriented software development which main goal is to isolate secondary or supporting functions of a program out of the main program's logic.

In the implementation of a program some concerns are scattered through the code, the point in which the implementation of a concern is tangled with the code of other parts of the program is called a *crosscut*, and the concern as a whole is called an aspect. On Figure 2.4 the representation of different modules and an aspect can be seen, while a module has its own logic, an aspect is divided in different pieces of different modules.

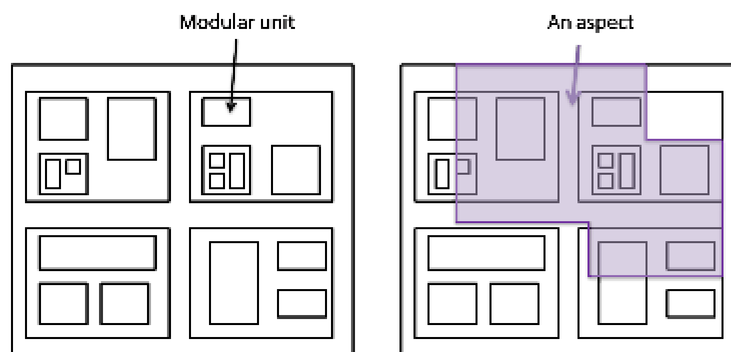

Figure 2.4: Example of module and aspect
Addapted from [17]

All aspect oriented programming implementations have some way to encapsulate all crosscutting expressions in one place, the difference between implementations is in the ability to define where a crosscut is needed. AspectJ [10] is a tool for AOP in Java.

2.4.1 AspectJ

AspectJ is an aspect-oriented extension for Java created at PARC (Palo Alto Research Center Incorporated). It is available as a plugging to the Eclipse IDE [26] or as a standalone program. AspectJ has become the standard for AOP and is one of the most powerful tool for AOP in Java.

A program in AspectJ has two components. The first one is the base code, which is a normal Java program. The second component are special constructs called *aspects*, they define the crosscutting behaviour of the aspect. An aspect in AspectJ has three key concepts:

Join point: Define particular points in the execution of a program, like method call, method return or variable access.

Pointcut: Allows to specify join points, an example of this can be seen on Listing 2.3, a pointcut is defined for the execution of any method beginning with *set* in class *C*.

```
1 pointcut set() : execution(* set*(..) ) && this(C);
```

Listing 2.3: Pointcut example

Advice: Defines the code to be added at the selected pointcut, an example can be seen on Listing 2.4, the code is executed after the pointcut occurs, it is possible to specify when the advice has to be executed.

```
1 after() : set() {
2   C.log();
3 }
```

Listing 2.4: Advice example

A programmer can also add new methods or fields to existing classes by defining them in the aspect, without the need to extend the class or change more code. For example, on Listing 2.5 a new method is added to class C.

```
1 void C.print() {
2   System.out.println(this);
3 }
```

Listing 2.5: Add a new method

Chapter 3

TXL (Turing eXtender Language)

Contents

3.1 Grammar definition **16**

3.1.1 Tokens 17

3.1.2 Compounds 18

3.1.3 Keys 18

3.1.4 Comments 18

3.1.5 Nonterminal definition 19

3.2 Transformation rules **20**

3.2.1 Parameters 21

3.2.2 Global variables 21

3.2.3 Constructor and deconstructors 21

3.2.4 Conditions 22

3.2.5 Conditional rules 23

A program transformation is any operation that takes the source code of a program and generates another program. These transformations can be done manually, but often it is more practical to use a tool that does them automatically. There are several tools that allow program transformations, some of them are no more than a programming language and a interpreter or compiler like Coccinelle [18], Stratego/XT [6] or TXL [7], others offer a full programming environment like DMS [3] or ASF+SDF [2].

In our case we have chosen to use TXL since it has a free implementation and it already has an updated Java grammar.

TXL, stands for Turing eXtender Language, is a programming language specifically designed for creating, manipulating and prototyping of new programming languages, but it can be used to apply any kind of transformation on source code [7,15,22]. It is a hybrid between functional and rule-based languages that uses first order functional programming at the higher level and term rewriting at the lower level.

TXL follows the same process as any other source transformation language when transforming code. First the input is read and parsed creating a parse tree of the source, then rules are applied to this tree and a new parse tree is created, and finally the transformed tree is unparsed to obtain the output text. This process can be seen on Figure 3.1, the string *"blue fish"* is used as input. First the parser creates a tree based on the grammar. Then the transformations are applied so that the words *blue fish* are transformed into the word *marlin* and finally the tree is unparsed getting the output string *"marlin"*.

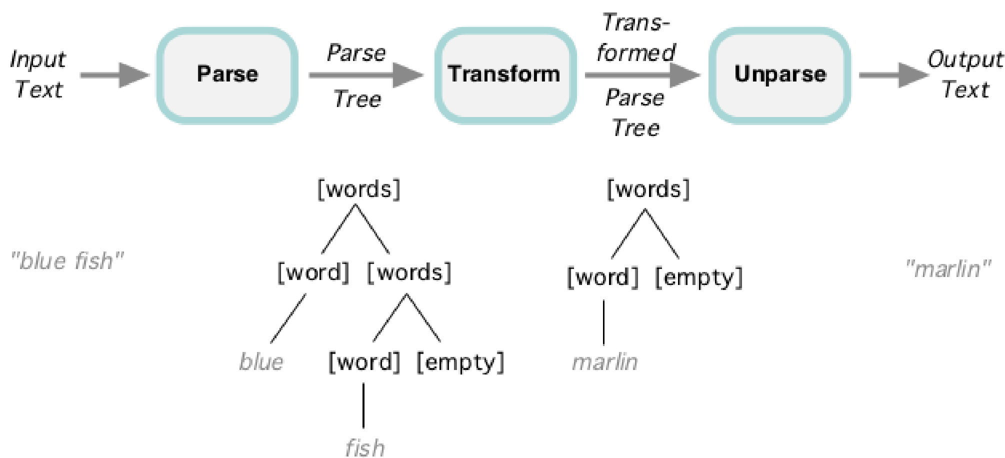

Figure 3.1: TXL process

A TXL program has two main components: the grammar and the transformation rules. The TXL processor interprets the code so the program components do not have to be compiled. This allows for easier maintenance and faster tests when developing.

3.1 Grammar definition

Grammar files define the lexical and syntactic forms of the input language. Usually the grammar is kept in a separate file and then included on the program using an `include`

statement. Grammar overrides can be used to redefine existing syntactic forms to allow new output or intermediate forms that are not in the original input language, this can make it easier to create transformations. A TXL grammar is composed of different elements.

3.1.1 Tokens

Tokens are the most basic elements, they define the basic, indivisible elements of a programming language, these elements are also called terminal symbols.

The `tokens` statement can be used to add new user defined tokens to the grammar or to modify existing ones. TXL has already some tokens defined, for example identifiers, numbers, character literals and strings. On Listings 3.1 a token is created defining an hexadecimal number.

```
1 tokens
2   hexnumber "0[Xx][\dABCDEFabcdef]+"
3 end tokens
```

Listing 3.1: TXL tokens

Regular expressions are used to define tokens and they will always try to match the longest possible sequence of input characters. Once an input item is matched into a type the item is permanently typed and its type cannot be changed. In cases of ambiguity, where more than one input token type may match an input item, the first matching type is used. User defined tokens have preference over the ones defined internally on TXL. If a name for a token is the name of a predefined type or a previously defined type the new definition will override the old one. An empty pattern can be used to undefine a type since it will not match with anything.

Several definitions can be added for a token using the vertical bar, `|`, for example on Listings 3.2, the hexnumber token has two definition for the x and X alternatives for an hexadecimal number.

```
1 tokens
2   hexnumber   "0X[\dABCDEFabcdef]+"
3               | "0x[\dABCDEFabcdef]+"
4 end tokens
```

Listing 3.2: TXL tokens, different definitions

A new definition can also be added to the token using an ellipsis, "...", that represents the previously defined definitions, then a vertical bar is added to include the new one. On Listings 3.3 an new definition is added to the `id` token to allow tokens that start with @ and & as the first character on a identifier.

```
1 tokens
2   id   ... | "[@&]\i*"
3 end tokens
```

Listing 3.3: TXL tokens, add definitions

3.1.2 Compounds

The compounds statement allows to define character sequences that need to go together, since TXL will treat special characters as separate ones, for example it would treat `<=` as two different symbols `<` and `=`. This does not matter in the transformation process, but it is used so that the output can be easier to read. Different compounds can be defined separated by a white space or a new line. An example of the compounds statement can be seen on Listings 3.4.

```
1 compounds
2   <= >= || &&
3 end compounds
```

Listing 3.4: TXL compounds

3.1.3 Keys

This statement allows some identifiers to be treated as keywords. Keywords are not matched by the `[id]` nonterminal and can only be matched by literal occurrences in a nonterminal definition or by the use of the predefined `[key]` nonterminal. As it can be seen on Listings 3.5 the key statement simply lists the identifiers that need to be treated as keywords.

```
1 keys
2   if while for
3 end keys
```

Listing 3.5: TXL keys

3.1.4 Comments

Comments can also be parsed, but they are ignored and treated as white space by default. An example of the use of the comments statement can be seen on Listings 3.6, if only one symbol is provided // on the example it means that the comment starts on the symbol and ends on the end of the line. If two symbols are on one line, /* */ on the example, it means that the comments start at the first symbol and ends on the second one.

```
1 comments
2   //
3   /* */
4 end comments
```

Listing 3.6: TXL comments

In order to parse the comments the TXL interpreter needs the *-comment* command line option to be used on execution this can also be overridden in the code using the "#**pragma** –comment" statement at the beginning of the program. In the case that comments are to be treated as input the grammar has to be able to handle them, if not the program will throw an error.

3.1.5 Nonterminal definition

Once all the terminal symbols are defined the nonterminal symbols or syntactic forms have to be defined. These are the ones that define the program structure, low level ones are composed of terminal symbols but higher level ones can be composed from other nonterminal or terminal symbols. We can see an example of a grammar definition for a Java method on Listing 3.7. The nonterminals are inside brackets so they are not treated as literals. Sequences are created using the `repeat` keyword, as shown on line 3 of Listings 3.7, and if the sequence contains elements separated by commas the `list` keyword can be used. The `opt` keyword is used to define when a part of the syntax is optional, for example on line 4 of Listings 3.7 the generic parameter is defined as optional for a Java method. When an element can have different definitions a bar (`|`) is used to separate them, this can be seen on Listings 3.8 where the type declaration for Java is defined, since Java can have classes, interfaces, or enumerations as types.

```
1 define method_declaration
2     [NL]
3     [repeat modifier]
4     [opt generic_parameter]
5     [type_specifier]
6     [method_declarator]
7     [opt throws]
8     [method_body]
9 end define
```

Listing 3.7: Java method grammar

```
1 define type_declaration
2         [class_declaration]       [NL][NL]
3     |   [interface_declaration]   [NL][NL]
4     |   [enum_declaration]        [NL]
5 end define
```

Listing 3.8: Java type grammar

All TXL programs start with the definition of the `[program]` nonterminal, since this is used to describe the structure of the entire input, and is used as the root of the parse tree.

Formatting cues are also implemented for pretty printing the output source. For example in Listing 3.7 line 2, `[NL]`, means that a new line is to be introduced at that point when printing the output. These cues do not affect the input parsing. Other cues are `[IN]` for an indent and `[EX]` for an exdent.

Base grammar is usually kept in a separate file and is never modified. Redefinitions modify a nonterminal in a grammar. They are the same as the definition of a nonterminal but with the `redefine` statement. A reference to the previous definition is made using an ellipsis as whith the tokens statement `"..."`, as shown on Listing 3.9, where an optional visited mark is added to a method.

3.2 Transformation rules

The other main part of a TXL program are the transformation rules. Rules define what changes are made to the input in order to obtain the desired output. Each rule defines a target type and a pattern, which is the part of the input that TXL is going to search for by pattern matching on the parse tree, and a replacement for that pattern. The transformation rules in TXL are strongly typed, this means that the replacement must be of the same type as the pattern, for example, if a pattern tries to match a method, the output has to be also a method, it cannot be a field. This can seem to be constraining, but that is what grammar redefinitions are for. An element can be redefined to accept otherwise illegal syntax so an specific transformation can be done, for example if we create a rule that visits different methods of the code and we do not want to revisit them we can create something as shown on Listing 3.9 so we know when each method has been visited.

```
1 redefine method_declaration
2   ... [opt visited_mark]
3 end redefine
```

Listing 3.9: Redefinition example

The pattern to match consists of a combination tokens and variables. To store something on a variable in a pattern first the name of the variable is written and then the type of the variable, for example the pattern `x [number] + y [number]` will try to find a place where two numbers are added in the input and it will store them in `x` and `y` variables. These variables can then be used, with other new nonterminal or tokens, in the replacement part of the rule to create the output parse tree.

Other rules can also be applied to variables in the replacement part of a rule by using postfix square bracket notation, `x [f]` means apply rule `f` to variable `x`. When a rule is applied to a variable, the value of the variable is the scope of the rule. A rule can only transform code that is inside its scope. As an example the rule on Listing 3.10 will replace all additions by subtractions.

```
1 rule replace_additions
2   replace [expression]
3     x [number] + y [number]
4   by
5     x − y
6 end rule
```

Listing 3.10: Replace additions by subtractions

All TXL programs start with the `main` rule, this is a special rule which will be automatically applied to the entire input at the start of the program. Often this rule is only a function that is used to call other rules.

Within TXL there are two ways to apply a transformation, using a *rule* or using a *function*, rules try to match their pattern repeatedly on their scope, apply the transformation and continue searching until no matches are found. Functions, unlike rules, do not search within the scope, they try to match their pattern to their entire scope, a searching function can be created using `replace *` instead of `replace`, these functions will only replace the first match of their pattern on the scope.

3.2.1 Parameters

Rules can receive one or more parameters, the syntax for parameters can be seen on Listings 3.11. The parameters a rule takes are defined after the rule name, first the name of the parameter is written and after that the type of the parameter. Once inside the rule parameter names may be used anywhere.

```
1 rule name parameter1 [type1] parameter2 [type2] ...
2   replace [type]
3     pattern
4   by
5     replacement
6 end rule
```

Listing 3.11: Rule with parameters

3.2.2 Global variables

Rules are able to import or export global variables, a global variable is bound when a rule exports them and they are accessed when a rule imports them. The type of a global variable is defined the first time it is exported and it must be the same in every subsequent export of a variable. The import and export syntax can be seen on Listings 3.12.

```
1 export varName [type]
2   replacement
3
4
5 import varName [type]
6   pattern
```

Listing 3.12: Import and export syntax

3.2.3 Constructor and deconstructors

Some statements often used on rules are the constructors and deconstructors. Deconstruct statements act like functions, matching the entire pattern to its scope, as functions they can also be made to search inside the scope by using `deconstruct * [type] variable`. Constructors allow partial results to be bound to variables, making it easier to see how a rule works.

The function on Listing 3.13 shows how constructors and deconstructors are used. First, in lines 2-3 an addition is bound to `add` variable, then this variable is `deconstructed` on line 4 to match the pattern on line 5 and the two numbers of the addition are bound to two variables, `x` and `y`. After this a new addition is `constructed` on line 6, and bound to a variable called `new_addition`, using the structure of line 7. Finally this new addition is used as the replacement. For example for an input of `2 + 3`, after the `deconstruct` on line 4 we obtain two variables `x = 2` and `y = 3`, and on the `construct` statement the new addition that is created is `3 + 2`.

```
1  function replace_addition
2    replace [addition]
3      add [addition]
4    deconstruct add
5      x [number] + y [number]
6    construct new_add [addition]
7      y + x
8    by
9      new_add
10 end function
```

Listing 3.13: Constructor and deconstructor example

3.2.4 Conditions

Conditions can also be added to a function or a rule using the `where` statement, this statement is used just before the `by` statement. The function or rule will only replace the input if the `where` statement return true. An example of the use of a where statement can be seen on Listings 3.14. [<] used on line 7 is a TXL built in function that will only match in this case if N1 < N2.

```
1  rule sort
2    replace [repeat number]
3      N1 [number]
4      N2 [number]
5      Rest [repeat number]
6    where
7      N1 [< N2]
8    by
9      N2 N1 Rest
10 end rule
```

Listing 3.14: Where statement example

There are 4 types of conditions in TXL:

- **where:** It will succeed if at least one of the conditions is true.
- **where not:** It will only succeed if none of the conditions match.
- **where all:** Will only succeed if all the conditions match their patterns.
- **where not all:** Will succeed if at least one of the conditions match.

An example of several conditions on a where can be seen on Listings 3.15, in this case the input will be replaced only if N1 < N2 and if N1 = N2.

```
1  where all
2    N1 [< N2] [= N2]
```

Listing 3.15: Where statement different conditions

3.2.5 Conditional rules

In order to be able to use rules on where clauses a special type of rule is required, called condition rules. These rules test for a match to their pattern and do nothing else, they do not change the input. They have the same syntax as normal rules, but the `replace` keyword is replaced with `match` and there is no `by` clause. One of these rules can be seen on Listings 3.16, this rule will only check to see if its scope is 2.

```
1 function isTwo
2   match [number]
3     2
4 end function
```

Listing 3.16: Match example

Normal transformation rules can also be used as condition rules by prepending a `?` to the rule name in the `where` clause. These rules will only match if the rule does some transformation. They are commonly used as a check, to only follow with the transformation if there is something left to transform.

Chapter 4

Patterns & Implementation

Contents

4.1 Rule application order **26**

4.2 Transformation rules **27**

4.2.1 Extract with hook 28

4.2.2 Extract blocks 31

4.2.3 Remove different feature class body declarations 32

4.2.4 Remove non marked class body declarations 33

4.2.5 Remove modifiers 34

4.2.6 Remove imports 35

4.2.7 Erase if empty 36

4.3 Result **36**

4.4 Statistics **38**

Extraction pattern	**TXL rule**
Addition at the beginning of the method Addition at the end of the method Addition anywhere with a hook method	Extract with hook
Overwrite method	Extract block
Move entire method Move field	Remove different feature class body declaration
Remove field acces level modifiers	Remove modifiers
Move entire class	Remove non featured class body declaration Erase class if empty
-	Remove imports

Table 4.1: Relation between extraction patterns and TXL rules

In order to extract the different features from source code, different rules have been created on TXL based on patterns observed when extracting features manually on previous works [25]. The relation between the patterns and the transformation rules created in TXL is shown on Table 4.1. Some patterns have been implemented on a single TXL transformation rule as a simplification, for example the patterns addition at the beginning or at the end of a method are subsets of addition anywhere with a hook method, so a single rule has been created for the three of them.

All these rules have to be applied to each source file, once for each defined feature in order to obtain all the different features. Each of the transformations is identified by a mark, in some cases depending on where the mark is, one or other rule can be applied. Source code in which features are to be extracted has to be marked correctly. When applying the rules a feature has to be specified, and the rules will be applied for only that feature, leaving the chosen feature's source on the output, an example on these marks can be seen on Listing 4.2, as it can be seen on lines 6-7 and 32-33 different marks can be applied to a single field or method. After the execution of each transformation rule the markers of the rule are erased so that the output can be directly compiled.

We wrote a program in Java that acts as a user interface for the transformation rules of TXL. The flow of the program is shown in Figure 4.1, the program takes the marked source files and the features that have to be extracted, then it calls TXL once for each file and feature combination. The program is also able to look in the source files to search for features, so the user does not have to write them all. Finally it saves the output files of each feature on a separate folder, so they can be composed using FeatureHouse.

The rules developed in TXL can be divided into two groups: the ones that remove pieces of code and the ones that modify the code. All rules that remove pieces of code follow the same procedure as the rule shown on Listing 4.1 a visual representation of this can be seen on Figure 4.2. In this example the element to remove is `stmt2`, to do that the rule first matches and bounds to variables `stmt2` and the statements that come after it after that it replaces the match with just the statements that come after `stmt2` removing it from the original tree.

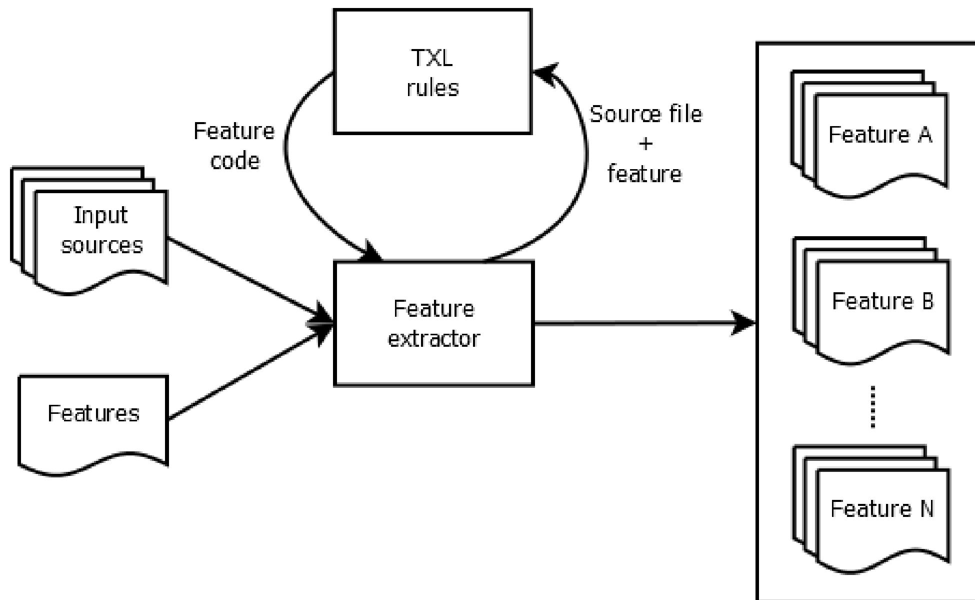

Figure 4.1: Program Flow

```
1 rule remove
2   replace [repeat statement]
3     rem  [element_to_remove]
4     rest [repeat statement]
5   by
6     rest
7 end rule
```

Listing 4.1: Remove elements

4.1 Rule application order

The transformation rules need an specific application order. The rules start from the innermost parts of the code and towards the outside, starting from method, constructor, ect. bodies and ending with classes.

1. Extract with hook.

   The first rule to be applied is the one that creates new methods with pieces of code selected from other methods. This is done because this method modifies the bodies of different class body declarations and creates new methods.

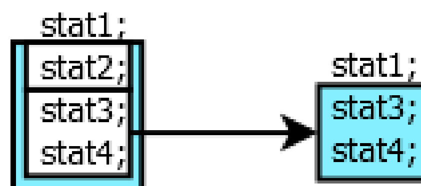

Figure 4.2: Statement removal example

2. Extract blocks.

   This rule can modify any previously defined method so it has to be applied second, since the first one creates new methods.

3. Remove non featured and different feature class body declarations.

   These two rules have to be applied in third place, they don't need an specific order between them, because the previous two rules can change the feature a class body declaration belongs.

4. Remove modifiers.

   This rule can be applied in any place, but at this moment the program is smaller since the previous rules have removed elements, so this rule will change less methods and fields, increasing the execution speed.

5. Remove imports.

   This rule can be applied in any order since its the only rule that modifies the imports on a Java file.

6. Erase if empty.

   This rule needs to be the last always. It checks if there are any empty classes after applying all the previous rules and removes them. After removing empty classes will also check if the file is empty and remove everything left since that Java file does not belong in the feature that is being extracted.

4.2 Transformation rules

Listing 4.2 will be used trough this section to explain the use of the rules.

```
1  import imp1.*;
2  #feature A
3  {
4    import imp2.*;
5  }
6
7  #feature C
8  public class C {
9    int i;
10   #feature A
11   double d;
12   #del_modif final
13   #feature B
14   final float f;
15
16   public C(){
17     stat1;
18     stat2;
```

```
19    #hook C, A
20          void stat3Hook()
21          stat3Hook();
22    {
23      stat3;
24    }
25  }
26
27  #feature A
28  public void m1(){
29    stat4;
30    #feature B
31    {
32      stat5;
33      stat6;
34    }
35    stat7;
36  }
37
38  #del_modif private, final
39  #feature A
40  private final void m2(){
41    stat8;
42    #hook A, B
43          void stat9Hook()
44          stat9Hook();
45    {
46      stat9;
47    }
48    stat10;
49  }
50
51  public void m3(){
52    stat11;
53    stat12;
54  }
55 }
```

Listing 4.2: Marked class

4.2.1 Extract with hook

This rule is used to extract a piece of code from a method or constructor that belongs to a feature, but the main method belong to another feature. The piece of code is extracted to a new method leaving a hook method call on the main one. Since this rule is a bit complicated the mark it uses needs more information as it can be seen on lines 19-24 or

42-47 in Listing 4.2. It needs two features, one for the main method and another one for the hook method, separated by a coma. Then it also needs the hook method's declaration and how it is going to be called from the main method. Below can be seen how the input from Listing 4.3 is transformed to the result on Listing 4.4, two different results are created when the feature to extract is the same as the main method's feature and when it is not. The empty hook method that can be seen on line 18 of Listing 4.4 is used as a place holder so the code can be compiled.

```
1  #feature 1
2  public void method(){
3    statement1();
4    #hook feature1, feature2
5          private int hook()
6          hook();
7    {
8      statement2();
9    }
10   statement3();
11 }
```

Listing 4.3: Hook input example

```
1  #feature feature1
2  public void method(){
3    statement1();
4    hook_call();
5    statement3();
6  }
7
8  //When the feature to extract
9  //is not "feature1"
10 #feature feature2
11 private void hook(){
12   statement2();
13 }
14
15 //When the feature to extract
16 //is "feature1"
17 #feature feature1
18 private void hook(){}
```

Listing 4.4: Hook creation

This rule's most important part can be seen on Listing 4.5. First a `class_body_declaration` is selected, this rule works for methods or constructors, in this case everything that comes after the method is also selected because on the replacement there has to be a new method for the hook, and since the input and output types of a rule on TXL have to be the same, to be able to have a list of methods, or

`class_body_declaration` in this case, on the output we need a list on the input. Using a constructor the new main method or constructor is created, simply replacing the block for the hook method by the hook call written on its extraction mark. On the other `construct` statement the hook method is created, using the block with the piece of code for the hook inside a new method is created with the block as the body and the hook method declaration defined on the extraction mark as its method declaration. An extraction mark is also added to this new method so it can later be extracted using a rule explained on section 4.2.3. Finally the original method or constructor is substituted by a modified one and the new hook method. The new hook method's body is erased and its mark's feature changed if the feature to extract is the same as the first feature defined on the hook extraction mark.

```
1  rule extract_with_hook
2    replace [repeat class_body_declaration]
3      decl [class_body_declaration]
4      rest [repeat class_body_declaration]
5    construct modified_decl
6                [class_body_declaration]
7      decl [replace_block_with_hook]
8    construct hook_method
9                [class_body_declaration]
10     decl [create_hook_method]
11   where
12     decl [?replace_block_with_hook]
13   by
14     modified_decl
15     hook_method
16     rest
17 end rule
```

Listing 4.5: Extract blocks with hook

The hook method, and previous method modification that is created from the marked block on lines 42-47 of Listing 4.2 can be seen on Listing 4.6. The block is replaced with the call to the hook method on line 6, the hook method is left empty for feature `A` on line 9-10 and on feature `B` which will later refine this method it has a body, as it can be seen on lines 13-16. The hook method created is also marked so it can be later extracted with other rule.

```
1  //for feature A
2  #del_modif private, final
3  #feature A
4  private final void m2(){
5    stat8;
6    stat9Hook();
7    stat10;
8  }
9  #feature A
10 void stat9Hook() {}
```

```
11
12 //for feature B
13 #feature B
14 void stat9Hook() {
15   stat9;
16 }
```

Listing 4.6: Extract blocks with hook result

In order to see the code for the whole transformation go to Appendix I.b.

4.2.2 Extract blocks

This rule is used when a feature overwrites a method or a constructor, the code that has to be on the method or constructor is written inside a block marked with a feature mark as can be seen on lines 30-34 of Listing 4.2. As it can be seen on the example the block can be anywhere on the method.

The main function of this rule can be seen on Listing 4.7. First other blocks that do not belong to the feature to extract are removed, only leaving the desired blocks an code that is not inside a block. Since the only code that has to be kept is inside the blocks, everything that is outside is erased. Once this is done the code is extracted from the blocks to the body of the method or constructor, and the method or constructor is marked for extraction with the same feature as the block, in order to be able to extract it with one of the rules explained on section 4.2.3 later.

```
1  function extract_blocks
2    replace [program]
3      p [program]
4    by
5      p [remove_different_feature_blocks]
6        [change_featured_block_method]
7        [change_featured_block_marked_method]
8        [change_featured_block_constructor]
9        [change_featured_block_marked_constructor]
10       [remove_block_feature_markers]
11 end function
```

Listing 4.7: Extract blocks

This rule is only applied if a block marked with a feature mark is found inside a method or a constructor, else it does not change anything.

For example the result of extracting the block that is on lines 30-34 of Listing 4.2, for feature `B`, can be seen on Listing 4.8. All the statements that were outside the block have been removed, and the feature mark of the method has been changed to `#feature B`, since the method now contains a body belonging to feature `B`.

```
1 #feature B
2 public void m1(){
3     stat6;
4 }
```

Listing 4.8: Extract blocks result example

To see the full code of the rule go to Appendix I.c.

4.2.3 Remove different feature class body declarations

This rule removes anything inside a class that is marked as belonging to other feature, field and methods are the most common ones. It uses the `#feature [feature_name]` extraction mark, as it can be seen on Listing 4.2 on lines 10-11 the field `double d` is marked as belonging to feature `A` and on lines 27-28 method `m1()` belongs also to feature `A`.

This rule in TXL is defined as seen on listing 4.9. It is composed of one function and two rules, one to erase the class body declarations whose features are different from the feature the rule is extracting and another one to remove all the marks on them. The function is used so the rules are applied only once in the order specified.

```
1  function remove_different_feature_cbd
2      replace [program]
3      p [program]
4    by
5      p [remove_different_f_cbd]
6        [remove_cbd_marks]
7  end function
8
9  rule remove_different_f_cbd
10   replace * [class_body_declaration*]
11     cbd [marked_class_body_declaration]
12     Rest [class_body_declaration*]
13   import TXLargs [repeat stringlit]
14   deconstruct * TXLargs
15     "-feature" f_name [stringlit]
16     more_options [repeat stringlit]
17   construct f_name_id [id]
18     _ [+ f_name]
19   deconstruct * [feature_mark] cbd
20     '#feature f_name_cbd [id]
21   where not
22     f_name_id [= f_name_cbd]
23   by
24     Rest
25 end rule
26
27 rule remove_cbd_marks
28   replace * [class_body_declaration]
29     cbd [marked_class_body_declaration]
30   deconstruct cbd
```

```
31      marks [repeat extraction_mark]
32      new_cbd [normal_class_body_declaration]
33    where
34      marks [?remove_feature_mark]
35    by
36      marks [remove_feature_mark]
37      new_cbd
38  end rule
```

Listing 4.9: Remove different feature class body declarations

The rule `remove_different_feature_cbd` follows the procedure shown at the beginning of this section, it searches for a `class_body_declaration` inside a the whole program. Once the element to delete is found, the entire structure captured with the `replace` statement is replaced with only the second element captured on the `by` statement of the rule, in this case a list containing various `class_body_declaration`. Doing this erases the selected `class_body_declaration` from the input, since the replacement is lacking the element to delete. But in this case the element is to be deleted only when the feature to extract and the element's feature are different. This is why the `when` statement is used, this statement checks if the rules it has inside are successfully applied and if they are it continues with the rule's replacement. In this case, a `not` keyword is also used to specify that it has to replace only if the rule fails to apply. The conditions uses the `[= ]` function, which is a built in TXL function that takes an argument and checks if it matches its scope, to compare two `[id]` type variables. These two variables are the method's feature name, which is obtained via a deconstruct that searches inside the `class_body_declaration` for a feature mark and binds the feature name to the `f_name_cbd` variable. The other feature name is obtained from the arguments passed to the program when executing it, this can be obtained inside a TXL program with the line `import TXLargs [repeat stringlit]`, then inside the arguments a structure that looks like `-feature [feature_name]` is searched to obtain the feature name passed to the program as argument, the name is bound to the `f_name` variable. More argumnets could be also passed to the program, but in this deconstruct they are ignored. After that the feature name is transformed to an `[id]` so it can be compared.

The second rule, `remove_cbd_marks`, uses the same process as explained before to remove the marks, it searches for a marked `class_body_declaration` and removes the feature mark from it.

4.2.4 Remove non marked class body declarations

These rule is used to remove elements without an extraction mark when the class belong to a feature that is not the one that the rules are currently extracting, leaving only elements relevant to the current feature. The elements themselves are not marked, but the class has to be marked as it is shown on line 7 of Listing 4.2, this way all the elements that are not marked belong to the same feature as the class.

On Listing 4.10 the main function form removing non featured methods is shown. This function is called once for each class of the file, this class is passed to the function as an argument. The feature to extract and the class' feature are obtained and 2 different rules

are applied to the class with these variables as arguments, these rules are only replace the class if the two arguments are different. The first rule, `remove_non_marked_cbd`, removes any element on the class that does not have a mark. And the second rule, `remove_non_f_cbd`, removes any class body declaration that does not have a feature mark, but it could have a mark to remove a modifier, for example.

```
1  function remove_non_featured_cbd_class
2            class [type_declaration]
3    replace [repeat type_declaration]
4      classes [repeat type_declaration]
5    deconstruct class
6      f_mark [feature_mark]
7      type_dcl [normal_type_declaration]
8    import TXLargs [repeat stringlit]
9    deconstruct * TXLargs
10     "-feature" f_name [stringlit]
11     more_options [repeat stringlit]
12   construct f_name_id [id]
13     _ [+ f_name]
14   deconstruct f_mark
15     '#feature f_name_class [id]
16   construct new_class [type_declaration]
17     f_mark
18     type_dcl [remove_non_marked_cbd
19                f_name_id f_name_class]
20              [remove_non_f_cbd
21                f_name_id f_name_class]
22   by
23     classes [. new_class]
24 end function
```

Listing 4.10: Remove non marked class body declarations

On the `by` part of this function the modified class is added to the scope, using the `[. ]` function to append the new class to the existing list of classes, creating again a list of the classes that are found on the input file.

To see the code of this transformation rule go to Appendix I.d.

4.2.5 Remove modifiers

This rule removes the defined modifiers from a method or a field. The mark it uses can be seen on lines 12 and 38 of Listing 4.2. It requires a list of different modifiers separated by commas that it has to delete.

The main rule for removing field modifiers can be seen on Listing 4.11. This rule searches for a marked field that has the delete modifier mark, if it does not have one the `deconstruct` statement fails and the rule does not replace anything. After removing the modifiers from the field is marked as applied, so TXL does not try to apply this rule again to the same field. This mark is removed once all modifiers are removed.

```
1  rule remove_f_modifiers
2    replace [marked_field_declaration]
3      mark [extraction_mark]
4      modif [repeat modifier]
5      type [type_specifier]
6      var [variable_declarators] ';
7    deconstruct * [delete_modifier_mark] mark
8      '#del_modif modifiers [list modifier]
9    by
10     mark [add_applied_mark]
11     modif [remove_modif each modifiers]
12     type
13     var ';
14 end rule
```

Listing 4.11: Remove field modifiers

Appendix I.f has all the code belonging to this rule.

4.2.6 Remove imports

This rule is used to erase `import` statements belonging to a different feature. The mark it uses can be seen on Listing 4.2, line 2-5. The main rule is shown on Listing 4.12, first the imports are searched for a block, then the block is deconstructed to obtain the imports inside. Finally the extracted import are erased if the block's feature and the feature to extract are different, if not, they are added to the rest of the imports.

```
1  rule remove_different_feature_imports
2    replace * [repeat
3              block_or_normal_import_declaration]
4      block_imp [block_import_declaration]
5      rest [repeat
6            block_or_normal_import_declaration]
7    import TXLargs [repeat stringlit]
8    deconstruct * TXLargs
9      "-feature" f_name [stringlit]
10     more_options [repeat stringlit]
11   construct f_name_id [id]
12     _ [+ f_name]
13   deconstruct block_imp
14     '#feature f_name_imp [id]
15     '{
16       imp [repeat
17            block_or_normal_import_declaration]
18     '}
19   by
20     imp [erase_if_different
```

```
21        f_name_imp f_name_id]
22    [. rest]
23 end rule
```

Listing 4.12: Remove imports

The whole code of this transformation rule is in Appendix I.g.

4.2.7 Erase if empty

This last rule is used to detect when a class is empty so it can be deleted and not be included on the chosen feature, its main function can be seen on Listing 4.13. The rule's process follows 3 steps, first the feature mark of each class is changed to the feature to extract, but only if the class is not empty. Then any class whose feature is different from the feature to extract is deleted. And finally it checks if the file is empty, if it is, the entire file is replaced by `//empty`.

```
1 function erase_if_empty
2   replace [program]
3     p [program]
4   by
5     p [change_marks]
6       [erase_if_other_feature]
7       [check_if_empty]
8 end function
```

Listing 4.13: Remove field modifiers

The TXL code of this transformation rule is on Appendix I.h.

4.3 Result

If these rules are applied to the class used as an example on Listing 4.2 the output can be seen on Listing 4.14. Class `C` only keeps the marked elements for each feature, and some of them are modified, field `f` that was declared on line 14 of Listing 4.2 now only appears on feature `B` and without the `final` modifier as it can be seen on line 26 of Listing 4.14. Hook methods are also created as defined, the hook method `stat9Hook()` first declared on line 19 for feature `A` and later refined by feature `B` on lines 31-33 of Listing 4.14 was first declared on lines 42-47 of Listing 4.2. All the non marked elements are moved to feature `C` since this was the class' feature.

```
1 //Feature A
2 import imp1.*;
3 import imp2.*;
4
5 public class C {
6     double d;
7     void stat3Hook () {
8         stat3;
```

```
9      }
10     public void m1 () {
11         stat4;
12         stat7;
13     }
14     void m2 () {
15         stat8;
16         stat9Hook ();
17         stat10;
18     }
19     void stat9Hook () {}
20 }
21
22 //Feature B
23 import imp1.*;
24
25 public class C {
26     float f;
27     public void m1 () {
28         stat5;
29         stat6;
30     }
31     void stat9Hook () {
32         stat9;
33     }
34 }
35
36 //Feature C
37 import imp1.*;
38
39 public class C {
40     int i;
41     public C () {
42         stat1;
43         stat2;
44         stat3Hook ();
45     }
46     void stat3Hook () {}
47     public void m3 () {
48         stat11;
49         stat12;
50     }
51 }
52
53 //Any other feature
54 //empty
```

Listing 4.14: Marked class result

4.4 Statistics

Table 4.2 shows the amount of elements that were implemented in TXL. On the left hand side of the table the new grammar definitions and changes needed are shown. On the right side the number of rules implemented and the amount of lines is shown.

Grammar changes	
New compounds	4
Symbol redefinitions	4
New symbols	15
Total lines	108

TXL implementation	
Rules/functions	52
Total lines	726

Table 4.2: TXL statistics

Chapter 5

Case studies

Contents

5.1 VOD Player Running Example **40**

5.1.1 Refactoring code criteria 41

5.1.2 Identifying features in code 41

5.2 ArgoUML **42**

5.3 Statistics **43**

In this chapter the two case studies that were used and their results will be explained.

5.1 VOD Player Running Example

Requirement/Feature	Requirement description	Refactoring Criteria
R1/SelectMovie	Display a list of movies and select one	Code that allows getting & displaying a list of movies and selecting one
R2/PlayImm	Play movie immediately after selection	Code enabling playing a movie immediately after it is selected
R3/Detail	Display textual movie information	Every code line, button, attribute that displays a movie detail frame
R4/Pause	Pause a movie	Code and buttons that enables a movie to be paused
R5/-	3 seconds max to load movie list	*Non-functional requirement*
R6/-	3 seconds max to load movie textual	*Non-functional requirement*
R7/-	1 second max to start playing a movie	*Non-functional requirement*
R8/VRCInterface	Provide VCR-like user interface	General graphic player UI. Dimensions of the frame, general buttons & labels
R9/StopMovie	Stop a movie	Piece of code & buttons that enables a movie to be stopped
R10/StartMovie	Start a movie	Code making possible watching a movie. Load movie
R11/ChangeServer	Change server	Code making possible a change of the server
R12/QuitPlayer	Exit the player	Code lines that enable closing and quitting the VOD Player
R13/StartPlayer	Start the movie player	The implementation concerning the running of the VOD Player

Table 5.1: VOD Player Requirements, Features, and Refactoring Criteria

VOD Player case study is a program that allows to watch video from a server, and it is written in Java, it has 42 classes of which 12 are listeners. The different features of our running example are defined on table 5.1. Requirements 5, 6 and 7 are non-functional requirements, meaning that they specify criteria to judge the operation of the system, rather than specific behaviours. Thus, these requirements are not taken as features for us, and they would not have their own module containing code. There are 4 classes where most of the features add their functionality. Those classes are Java frames, we give a short description below.

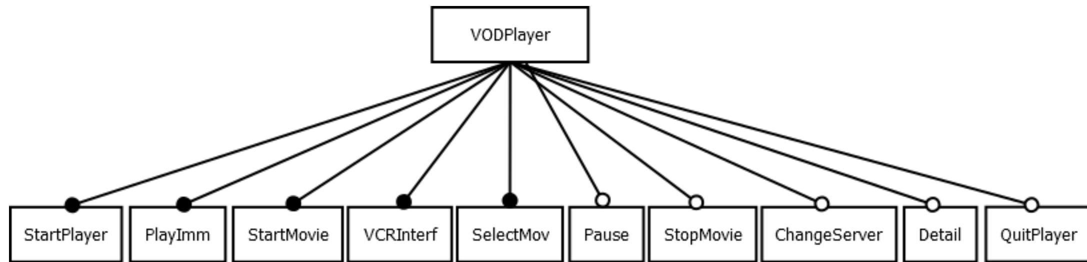

Figure 5.1: VODPlayer feature model

- `VODClient`: is the main frame of the application and when a user connects to `VODPlayer` this frame is displayed and covers all the features listed above.
- `ListFrame`: is the frame that is displayed after clicking the `Movies` button of `VODClient` frame. At this point the user can connect to another server, select a movie, display its details or close the frame.
- `Detail`: this frame is displayed after clicking `Detail` button on `ListFrame` frame, displaying information about the selected movie.
- `ChangeServer`: this frame is displayed after clicking `Servers` button on `ListFrame` frame which allows users to connect to another server to get a new movie list.

5.1.1 Refactoring code criteria

We defined the description and functionality of each feature conceptually. But it is not enough, we should define as well which elements at code level should implement a feature. This criteria specifies whether a feature implements a code element or not. Table 5.1 shows the criteria used for the VOD player case study.

5.1.2 Identifying features in code

When developing features, two kind of features can be found, optional or mandatory ones. Optional features are the ones that can be removed without the program loosing its ability to execute properly. Mandatory features on the other hands are necessary for the execution of the program and need to be included in some, if not all, variations of a program.

We define which feature are either mandatory or optional. We introduced optional features, to give variability to the model and to see what kind of issues does the optional feature causes.

- Mandatory features: *StartPlayer*, *PlayImm*, *StartMovie*, *VCRInterf* and *SelectMov*. These features are always included in the composition process.
- Optional features: *QuitPlayer*, *Pause*, *StopMovie*, *ChangeServ* and *Detail*. These features may or may not be included in the final configuration of the product. These features are the ones that give variability to the model.

The feature model of this SPL can be seen on Figure 5.1. The 5 optional features gives us 32 variants on this software product line.

5.2 ArgoUML

ArgoUML [1] is an open source UML modelling tool written in Java. For our case study we used a version of ArgoUML in which it was already tried to extract the different features in order to create a SPL using a Java preprocessor called Javapp [4,16]. What we had to do is transform the annotations so they could work with our transformation rules. The features and their description of ArgoUML are shown on Table 5.2.

Requirement/feature	**Requirement description**	**Refactoring criteria**
R1/ClassDiagram	Allows for the creation of class diagrams	Code that allows the creation and handling of class diagrams
R2/ActivityDiagram	Allows for the creation of activity diagrams	Code that allows the creation and handling of activity diagrams
R3/StateDiagram	Allows for the creation of state diagrams	Code that allows the creation and handling of state diagrams
R4/SequenceDiagram	Allows for the creation of sequence diagrams	Code that allows the creation and handling of sequence diagrams
R5/UseCaseDiagram	Allows for the creation of use case diagrams	Code that allows the creation and handling of use case diagrams
R6/CollaborationDiagram	Allows for the creation of collaboration diagrams	Code that allows the creation and handling of collaboration diagrams
R7/DeploymentDiagram	Allows for the creation of deployment diagrams	Code that allows the creation and handling of deployment diagrams
R8/Logging	Allows for the logging of actions	Code that logs events
R9/Cognitive	Gives cognitive support when creating diagrams	Code that gives cognitive support

Table 5.2: ArgoUML Requirements, Features, and Refactoring Criteria

This program has 1688 files and there are about 120KLoC in the implementation. The feature model of ArgoUML can be seen on Figure 5.2. The program has 1 mandatory feature which is the ability to draw class diagrams and 8 optional features, the ability to draw different UML diagrams, cognitive support for the diagrams and the logging.

This SPL also has another complication that VODPlayer did not have, it has 16 interactions between features. Most of these interactions occur when two optional features are included, for example one of the optional diagrams and the logging feature. For our transformations these interactions count as separate features, so the final feature list contains 25 different features that are:

- core
- cognitive
- activityDiagram
- stateDiagram

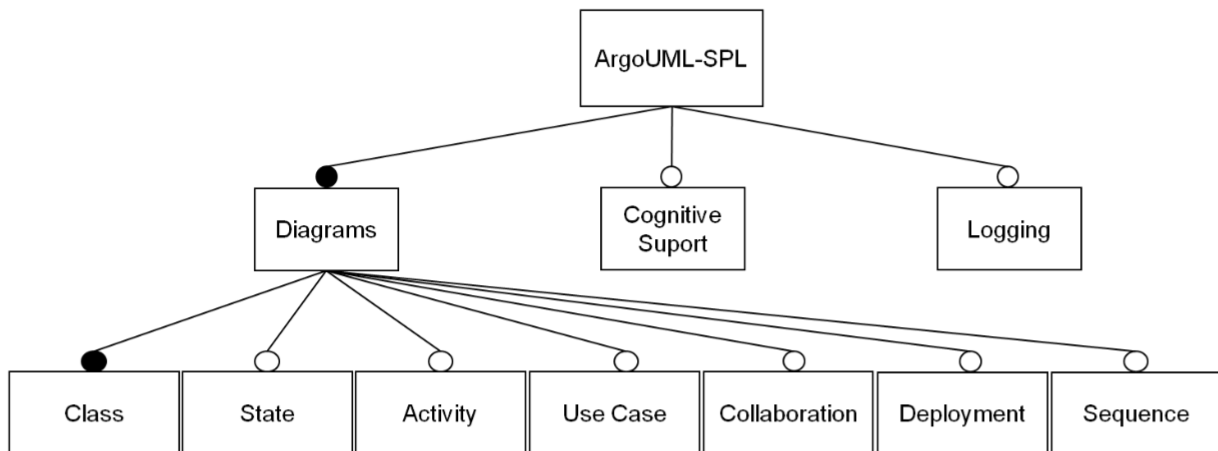

Figure 5.2: ArgoUML feature model as seen in [16]

- sequenceDiagram
- useCaseDiagram
- collaborationDiagram
- deploymentDiagram
- coreLogging
- cognitiveLogging
- activityDiagramLogging
- stateDiagramLogging
- sequenceDiagramLogging
- useCaseDiagramLogging
- collaborationDiagramLogging
- deploymentDiagramLogging
- stateOrActivityDiagramLogging
- cognitiveAndDeployment
- cognitiveAndSequence
- activityAndState
- deploymentAndUse
- collaborationAndSequence
- stateOrActivityDiagram
- collaborationOrSequenceDiagram
- collaborationOrActivityDiagram

When marking this program we realized that `import` statements could also belong to different features. We had to develop another transformation rule in order to be able to extract the import statements. A few changes were also needed on other rules to handle nested classes and enumerations.

5.3 Statistics

In each of the program different kinds of annotations had to be made, the number and type of the annotations can be seen on Table 5.3. A comparison between types of annotations can also be seen on Figure 5.3.

Annotated elements	VODPlayer	ArgoUML
Import blocks	0	374
Type declarations	42	1770
Class body declarations	27	326
Remove modifiers	8	0
Blocks	4	95
Blocks for hook	13	1040

Table 5.3: Annotation statistics

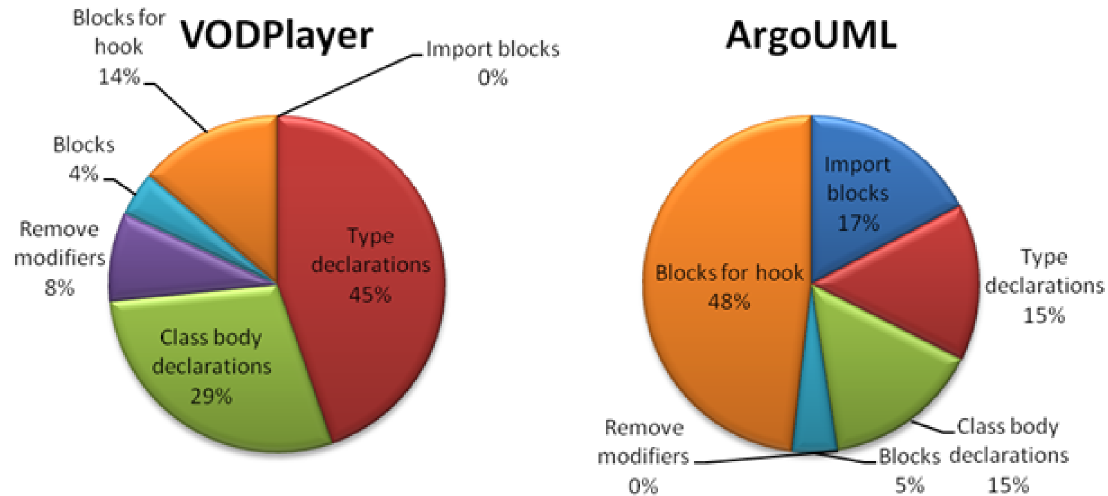

Figure 5.3: Annotations statistics comparison

Besides the annotations, for ArgoUML, 62 new methods had to be created in order to extract some parts of code. For example, with our rules, we cannot extract elements that are inside the condition on an `if` statement, in these cases we had to create a method that would return false if the when the feature is not included in the program and the required element when the feature is included.

The execution time data is shown on Table 5.4. The execution time increases linearly with the number of files and features to extract.

	VODPlayer	**ArgoUML**
Total	15.6s	1240.2s
Mean	33.61ms	29.28ms
Standard deviation	20.29ms	17.2ms
Median	31ms	31ms
Files	42	1688
Features	10	25

Table 5.4: Execution time statistics

Chapter 6

Conclusions and future work

Contents

6.1 Conclusions **46**
6.1.1 Implementation 46
6.1.2 Annotation 46
6.1.3 Viability of TXL 46
6.2 Future work **46**

6.1 Conclusions

This project is based on the FOSD paradigm, more specifically on feature extraction from an already composed program. For this goal we developed seven transformation rules in TXL [7] from eight extraction pattern that were already defined. To ensure that these rules work we tested with two programs, VODPlayer and ArgoUML. We also used FeatureHouse [9] in order to compose the program with all the variants once we had all the features extracted. Due to time constraints we could not automate the creation of the variations for ArgoUML and the test for this program were done by hand and using only some of the most significant variations. The annotation of ArgoUML took four weeks, the compilation of the program, since it uses different source folders and then applies some transformations that change the package structure of the program, took more time than expected to understand and modify.

6.1.1 Implementation

Seven transformation rules have been implemented to allow for feature extraction. A way to annotate source code in order to be able to apply the transformation rules is also provided.

A program has also been written in Java to handle the calls to the transformation rules for each file and feature combination, and work as a user interface.

6.1.2 Annotation

We have also noticed that large programs are very difficult and time consuming to annotate, and not only for the size they have. In large programs there are several coding styles and the annotations have to be adapted to the program, sometimes changing the structure of an element in order to be able to annotate the part that is going to be extracted. The creation of hook methods is also complicated if the hook method is going to contain a large piece of code that need a lot of arguments to be passed to the hook method. In this case we found very difficult to keep track of all the arguments needed.

6.1.3 Viability of TXL

We have also found that TXL is very powerful tool that allows for very complicated things, from little changes to the code to translating a program that is written in a language to a new one. This is also the reason that TXL has a very steep learning curve. A person that has not worked with functional programming before will have a very hard time adjusting to the way TXL works.

6.2 Future work

We propose the integration of the annotations an the transformation rule on an IDE, Eclipse for example. This integration will allow for easier annotations. Another improvement that could also be done is on the creation of hook methods, now when annotating a piece of code as belonging to a hook method the programmer has to define the parameter

that the hook method will need, this could be done easier if it could be automatized by checking the piece of code to extract for variables it uses and defining which of them are not fields.

Right now an annotated program cannot be compiled, some improvements can be done to the TXL grammar for Java comment in order to allow the annotations to be inside the comments. With these changes comments could also be extracted with the code.

We also propose another way to annotate code that could not be done due to time constraints. When annotating a field, for example, as belonging to a feature all the pieces of code where that field is used could also be automatically included as belonging to that feature. Other simpler way to do it if the annotations and the transformation rules were integrated on an IDE would be to suggest pieces of code, that belong to an element already annotated, as belonging to that same feature. This will make it easier to find code that belongs to a feature.

Appendix I

TXL source code

Contents

I.a Main file 49
I.b Extract block with hook 50
I.c Extract block 52
I.d Remove non featured class body declarations 55
I.e Remove different feature class body declarations 56
I.f Remove modifiers 57
I.g Remove imports 59
I.h Erase if empty 59
I.i Grammar overrides 61

In this appendix the source code for the TXL transformation rules is provided.

I.a Main file

This file is the main file of the TXL transformation rules, it has the main function of the program an some general functions that are used through the different transformations.

```
include "java.grm"
include "featureOverrides.grm"

include "extract_block.txl"
include "extract_blocks_with_hook.txl"
include "remove_different_feature_cbd.txl"
include "remove_non_featured_cbd.txl"
include "remove_modifiers.txl"
include "remove_imports.txl"
include "erase_if_empty.txl"

function main
    replace [program]
    p [program]
  by
    p [extract_blocks_with_hook]
      [extract_blocks]
      [remove_non_featured_cbd]
      [remove_different_feature_cbd]
      [remove_empty_classes]
      [remove_modifiers]
      [remove_imports]
      [erase_if_empty]
      [remove_class_mark]
end function

rule remove_empty_classes
  replace * [class_body_declaration*]
    t [class_declaration]
    Rest [class_body_declaration*]
  where not
    t [?has_class_body_declaration]
  by
    Rest
end rule

function has_enum_element
  replace * [enum_element]
    e [enum_element]
  by
    e
end function

function remove_feature_mark
  replace * [repeat extraction_mark]
    feature_m [feature_mark]
    rest [repeat extraction_mark]
  by
    rest
end function

function create_feature_mark
  replace [repeat extraction_mark]
    marks [repeat extraction_mark]
  deconstruct not * [feature_mark] marks
    '#feature _ [id]
  import TXLargs [repeat stringlit]
  deconstruct * TXLargs
    "-feature" f_name [stringlit] more_options [repeat stringlit]
  construct f_name_id [id]
```

```
61      _ [+ f_name]
62    construct new_mark [extraction_mark]
63      '#feature f_name_id
64    by
65      marks [. new_mark]
66  end function
67
68  function change_feature new_feature [id]
69    replace * [feature_mark]
70      '#feature _ [id]
71    by
72      '#feature new_feature
73  end function
74
75  rule remove_class_mark
76      replace * [type_declaration]
77      _ [feature_mark]
78      type_dcl [normal_type_declaration]
79    by
80      type_dcl
81  end rule
```

I.b Extract block with hook

This file contains the code belonging to the rule to extract a piece of code creating a hook method.

```
1   function extract_blocks_with_hook
2       replace [program]
3       p [program]
4     by
5       p [separate_classes]
6   end function
7
8   function separate_classes
9     replace * [repeat type_declaration]
10      dcl [repeat type_declaration]
11    by
12      _ [extract_hook_class each dcl]
13  end function
14
15  function extract_hook_class dcl [type_declaration]
16    replace [repeat type_declaration]
17      t [repeat type_declaration]
18    construct new_dcl [type_declaration]
19      dcl [extract_with_hook2]
20    by
21      t [. new_dcl]
22  end function
23
24  function extract_with_hook2
25    replace * [repeat class_body_declaration]
26      dcl [repeat class_body_declaration]
27    by
28      _ [apply_rule each dcl]
29  end function
30
31  function apply_rule dcl [class_body_declaration]
32    replace [repeat class_body_declaration]
33      dcls [repeat class_body_declaration]
34    construct dcl_list [repeat class_body_declaration]
35      dcl
36    construct new_dcl [repeat class_body_declaration]
37      dcl_list [extract_with_hook]
38               [extract_with_hook_class]
39    by
```

```
40      dcls [. new_dcl]
41  end function
42
43  function extract_with_hook_class
44    replace [repeat class_body_declaration]
45      decl [class_body_declaration]
46      rest [repeat class_body_declaration]
47    deconstruct * [type_declaration] decl
48      type [type_declaration]
49    construct new_dcl [class_body_declaration]
50      type [extract_with_hook2]
51    by
52      new_dcl
53      rest
54  end function
55
56  rule extract_with_hook
57    replace [repeat class_body_declaration]
58      decl [class_body_declaration]
59      rest [repeat class_body_declaration]
60    deconstruct not decl
61      _ [type_declaration]
62    construct modified_decl [class_body_declaration]
63      decl [replace_block_with_hook]
64    construct hook_method [class_body_declaration]
65      decl [create_hook_method]
66    where
67      decl [?replace_block_with_hook]
68    by
69      modified_decl
70      hook_method
71      rest
72  end rule
73
74  function replace_block_with_hook
75    replace * [repeat declaration_or_statement]
76      b [marked_block]
77      rest [repeat declaration_or_statement]
78    deconstruct not b
79      _ [feature_mark]
80      _ [block]
81    deconstruct * [hook_method_mark] b
82      '#hook f_name1 [feature_name] ', f_name2 [feature_name]
83      hook_d [hook_declaration]
84      hook_c [hook_call]
85    deconstruct hook_c
86      dcl_or_sta_hook_c [declaration_or_statement]
87    by
88      dcl_or_sta_hook_c
89      rest
90  end function
91
92  function create_hook_method
93    replace [class_body_declaration]
94      decl [class_body_declaration]
95    deconstruct * [block] decl
96      body [block]
97    deconstruct body
98      '{
99        statements [repeat declaration_or_statement]
100     '}
101   deconstruct * [marked_block] statements
102     mb [marked_block]
103   deconstruct mb
104     _ [repeat extraction_mark+]
105     new_body [block]
106   deconstruct * [hook_method_mark] mb
107     '#hook f_name1 [feature_name] ', f_name2 [feature_name]
108     hook_d [hook_declaration]
109     _ [hook_call]
```

```
110  deconstruct hook_d
111    modif [repeat modifier]
112    gen_param [opt generic_parameter]
113    type [type_specifier]
114    declarator [method_declarator]
115    throws [opt throws]
116  construct mark [feature_mark]
117    '#feature f_name2
118  construct hook_method [marked_class_body_declaration]
119    mark
120    modif
121    gen_param
122    type
123    declarator
124    throws
125    new_body
126  by
127    hook_method [erase_if_original f_name1]
128 end function
129
130 function erase_if_original original_f [feature_name]
131  replace [marked_class_body_declaration]
132    _ [feature_mark]
133    modif [repeat modifier]
134    gen_param [opt generic_parameter]
135    type [type_specifier]
136    declarator [method_declarator]
137    throw [opt throws]
138    _ [method_body]
139  import TXLargs [repeat stringlit]
140  deconstruct * TXLargs
141    "-feature" f_name [stringlit] more_options [repeat stringlit]
142  construct f_name_id [id]
143    _ [+ f_name]
144  construct new_body [method_body]
145    '{ '}
146  deconstruct original_f
147    f_name_orig [id]
148  construct new_mark [feature_mark]
149    '#feature original_f
150  where
151    f_name_id [= f_name_orig]
152  by
153    new_mark
154    modif
155    gen_param
156    type
157    declarator
158    throw
159    new_body
160 end function
```

I.c Extract block

This file contains the code that is used to extract marked block so latter they can be used to overwrite pieces of code when composing the program.

```
1
2 function extract_blocks
3  replace [program]
4    p [program]
5  by
6    p [remove_different_feature_blocks]
7      [change_featured_block_method]
8      [change_featured_block_marked_method]
9      [change_featured_block_constructor]
```

```
10        [change_featured_block_marked_constructor]
11        [remove_block_feature_markers]
12  end function
13
14  rule remove_block_feature_markers
15    replace [repeat declaration_or_statement]
16      mb [marked_block]
17      rest [repeat declaration_or_statement]
18    deconstruct mb
19      mark [repeat extraction_mark]
20      b [block]
21    deconstruct b
22      '{
23        stats [repeat declaration_or_statement]
24      '}
25    by
26      stats [. rest]
27  end rule
28
29  rule remove_different_feature_blocks
30    replace [repeat declaration_or_statement]
31      b [marked_block]
32      rest [repeat declaration_or_statement]
33    import TXLargs [repeat stringlit]
34    deconstruct * TXLargs
35      "-feature" f_name [stringlit] more_options [repeat stringlit]
36    construct f_name_id [id]
37      _ [+ f_name]
38    deconstruct b
39      marks [repeat extraction_mark]
40      _ [block]
41    deconstruct * [feature_mark] marks
42      '#feature f_name2 [id]
43    where not
44      f_name_id [= f_name2]
45    by
46      rest
47  end rule
48
49  rule change_featured_block_method
50    replace * [class_body_declaration]
51      modif [repeat modifier]
52      gen_param [opt generic_parameter]
53      type [type_specifier]
54      declarator [method_declarator]
55      throws [opt throws]
56      body [method_body]
57    deconstruct body
58      '{
59        statements [repeat declaration_or_statement]
60      '}
61    construct new_body [method_body]
62      '{
63        statements [remove_non_block_parts]
64      '}
65    import TXLargs [repeat stringlit]
66    deconstruct * TXLargs
67      "-feature" f_name [stringlit] more_options [repeat stringlit]
68    construct f_name_id [id]
69      _ [+ f_name]
70    construct marks [repeat extraction_mark]
71      %empty
72    where all
73      statements [?has_this_feature_block] [?remove_non_block_parts]
74    by
75      marks [create_feature_mark] [change_feature f_name_id]
76      modif
77      gen_param
78      type
79      declarator
```

```
80     throws
81     new_body
82   end rule
83 
84   rule change_featured_block_marked_method
85     replace * [class_body_declaration]
86       marks [repeat extraction_mark]
87       modif [repeat modifier]
88       gen_param [opt generic_parameter]
89       type [type_specifier]
90       declarator [method_declarator]
91       throws [opt throws]
92       body [method_body]
93     deconstruct body
94       '{
95         statements [repeat declaration_or_statement]
96       '}
97     construct new_body [method_body]
98       '{
99         statements [remove_non_block_parts]
100      '}
101    import TXLargs [repeat stringlit]
102    deconstruct * TXLargs
103      "-feature" f_name [stringlit] more_options [repeat stringlit]
104    construct f_name_id [id]
105      _ [+ f_name]
106    where all
107      statements [?has_this_feature_block] [?remove_non_block_parts]
108    by
109      marks [create_feature_mark] [change_feature f_name_id]
110      modif
111      gen_param
112      type
113      declarator
114      throws
115      new_body
116  end rule
117 
118  rule change_featured_block_constructor
119    replace * [class_body_declaration]
120      modif [repeat modifier]
121      gen_param [opt generic_parameter]
122      declarator [constructor_declarator]
123      throws [opt throws]
124      body [constructor_body]
125    deconstruct body
126      '{
127        statements [repeat declaration_or_statement]
128      '}
129    construct new_body [constructor_body]
130      '{
131        statements [remove_non_block_parts]
132      '}
133    import TXLargs [repeat stringlit]
134    deconstruct * TXLargs
135      "-feature" f_name [stringlit] more_options [repeat stringlit]
136    construct f_name_id [id]
137      _ [+ f_name]
138    construct marks [repeat extraction_mark]
139      %empty
140    where all
141      statements [?has_this_feature_block] [?remove_non_block_parts]
142    by
143      marks [create_feature_mark] [change_feature f_name_id]
144      modif
145      gen_param
146      declarator
147      throws
148      new_body
149  end rule
```

```
150
151 rule change_featured_block_marked_constructor
152   replace * [marked_class_body_declaration]
153     marks [repeat extraction_mark]
154     modif [repeat modifier]
155     gen_param [opt generic_parameter]
156     declarator [constructor_declarator]
157     throws [opt throws]
158     body [constructor_body]
159   deconstruct body
160     '{
161       statements [repeat declaration_or_statement]
162     '}
163   construct new_body [constructor_body]
164     '{
165       statements [remove_non_block_parts]
166     '}
167   import TXLargs [repeat stringlit]
168   deconstruct * TXLargs
169     "-feature" f_name [stringlit] more_options [repeat stringlit]
170   construct f_name_id [id]
171     _ [+ f_name]
172   where all
173     statements [?has_this_feature_block] [?remove_non_block_parts]
174   by
175     marks [create_feature_mark] [change_feature f_name_id]
176     modif
177     gen_param
178     declarator
179     throws
180     new_body
181 end rule
182
183 rule has_this_feature_block
184   replace [repeat declaration_or_statement]
185     b [marked_block]
186     rest [repeat declaration_or_statement]
187   import TXLargs [repeat stringlit]
188   deconstruct * TXLargs
189     "-feature" f_name [stringlit] more_options [repeat stringlit]
190   construct f_name_id [id]
191     _ [+ f_name]
192   deconstruct * [feature_mark] b
193     '#feature f_name2 [id]
194   where
195     f_name_id [= f_name2]
196   by
197     rest
198 end rule
199
200 rule remove_non_block_parts
201   skipping [block]
202   replace [repeat declaration_or_statement]
203     stat [declaration_or_statement]
204     rest [repeat declaration_or_statement]
205   deconstruct not * [feature_mark] stat
206     m [feature_mark]
207   by
208     rest
209 end rule
```

I.d Remove non featured class body declarations

This file contains the rules that need to be applied in order to remove the non featured class body declarations from a Java file.

```
1  function remove_non_featured_cbd
2    replace * [repeat type_declaration]
3      classes [repeat type_declaration]
4    by
5      _ [remove_non_featured_cbd_class each classes]
6  end function
7
8  function remove_non_featured_cbd_class class [type_declaration]
9    replace [repeat type_declaration]
10     classes [repeat type_declaration]
11     deconstruct class
12     f_mark [feature_mark]
13     type_dcl [normal_type_declaration]
14   import TXLargs [repeat stringlit]
15   deconstruct * TXLargs
16     "-feature" f_name [stringlit] more_options [repeat stringlit]
17   construct f_name_id [id]
18     _ [+ f_name]
19   deconstruct f_mark
20     '#feature f_name_class [id]
21   construct new_class [type_declaration]
22     f_mark
23     type_dcl [remove_non_marked_cbd f_name_id f_name_class]
24            [remove_non_f_cbd f_name_id f_name_class]
25   by
26     classes [. new_class]
27 end function
28
29 rule remove_non_marked_cbd f_name_id [id] f_name_class [id]
30   replace * [class_body_declaration*]
31     cbd [normal_class_body_declaration]
32     Rest [class_body_declaration*]
33   deconstruct not cbd
34     _ [class_declaration]
35   deconstruct not cbd
36     _ [interface_declaration]
37   where not
38     f_name_id [= f_name_class]
39   by
40     Rest
41 end rule
42
43 rule remove_non_f_cbd f_name_id [id] f_name_class [id]
44   replace * [class_body_declaration*]
45     m_cbd [marked_class_body_declaration]
46     Rest [class_body_declaration*]
47   deconstruct not * [feature_mark] m_cbd
48     '#feature _ [id]
49   where not
50     f_name_id [= f_name_class]
51   by
52     Rest
53 end rule
```

I.e Remove different feature class body declarations

This file contains the rules used to remove class body declarations that belong to a feature that is not the one that is being extracted.

```
1  function remove_different_feature_cbd
2      replace [program]
3      p [program]
4    by
5      p [remove_different_f_cbd]
6        [remove_cbd_marks]
7  end function
```

```
8
9  rule remove_different_f_cbd
10   replace * [class_body_declaration*]
11     cbd [marked_class_body_declaration]
12     Rest [class_body_declaration*]
13   import TXLargs [repeat stringlit]
14   deconstruct * TXLargs
15     "-feature" f_name [stringlit] more_options [repeat stringlit]
16   construct f_name_id [id]
17     _ [+ f_name]
18   deconstruct * [feature_mark] cbd
19     '#feature f_name2 [id]
20   where not
21     f_name_id [= f_name2]
22   by
23     Rest
24 end rule
25
26 rule remove_cbd_marks
27   replace * [class_body_declaration]
28     cbd [marked_class_body_declaration]
29   deconstruct cbd
30     marks [repeat extraction_mark]
31     new_cbd [normal_class_body_declaration]
32   where
33     marks [?remove_feature_mark]
34   by
35     marks [remove_feature_mark]
36     new_cbd
37 end rule
```

I.f Remove modifiers

This file contains the rules used to remove the selected modifiers from method and fields.

```
1  function remove_modifiers
2      replace [program]
3      p [program]
4    by
5      p [remove_method_modifiers]
6        [remove_field_modifiers]
7  end function
8
9  function add_applied_mark
10   replace * [delete_modifier_mark]
11     '#del_modif modifiers [list modifier]
12   by
13     '#del_modif modifiers 'applied
14 end function
15
16 rule remove_modif modif [modifier]
17   replace [repeat modifier]
18     modif
19     rest [repeat modifier]
20   by
21     rest
22 end rule
23
24
25
26 function remove_field_modifiers
27     replace [program]
28     p [program]
29   by
30     p [remove_f_modifiers]
31       [remove_f_modifier_marks]
32 end function
```

```
33
34  rule remove_f_modifiers
35    replace * [marked_class_body_declaration]
36      mark [extraction_mark]
37      modif [repeat modifier]
38      type [type_specifier]
39      var [variable_declarators] ';
40    deconstruct * [delete_modifier_mark] mark
41      '#del_modif modifiers [list modifier]
42    by
43      mark [add_applied_mark]
44      modif [remove_modif each modifiers]
45      type
46      var ';
47  end rule
48
49  rule remove_f_modifier_marks
50    replace * [class_body_declaration]
51      mark [extraction_mark]
52      modif [repeat modifier]
53      type [type_specifier]
54      var [variable_declarators] ';
55    by
56      modif
57      type
58      var ';
59  end rule
60
61
62  function remove_method_modifiers
63      replace [program]
64      p [program]
65    by
66      p [remove_m_modifiers]
67        [remove_m_modifier_marks]
68  end function
69
70  rule remove_m_modifiers
71    replace * [marked_class_body_declaration]
72      mark [extraction_mark]
73      modif [repeat modifier]
74      gen_param [opt generic_parameter]
75      type [type_specifier]
76      declarator [method_declarator]
77      throws [opt throws]
78      body [method_body]
79    deconstruct * [delete_modifier_mark] mark
80      '#del_modif modifiers [list modifier]
81    by
82      mark [add_applied_mark]
83      modif [remove_modif each modifiers]
84      gen_param
85      type
86      declarator
87      throws
88      body
89  end rule
90
91  rule remove_m_modifier_marks
92    replace * [class_body_declaration]
93      mark [extraction_mark]
94      modif [repeat modifier]
95      gen_param [opt generic_parameter]
96      type [type_specifier]
97      declarator [method_declarator]
98      throws [opt throws]
99      body [method_body]
100   by
101     modif
102     gen_param
```

```
103      type
104      declarator
105      throws
106      body
107  end rule
```

I.g Remove imports

The next file contains the required rules to remove imports that do not belong in the feature that is being extracted.

```
1   function remove_imports
2       replace * [repeat block_or_normal_import_declaration]
3       imp_dcl  [repeat block_or_normal_import_declaration]
4     by
5       imp_dcl [remove_different_feature_imports]
6   end function
7
8   rule remove_different_feature_imports
9     replace * [repeat block_or_normal_import_declaration]
10      block_imp [block_import_declaration]
11      rest [repeat block_or_normal_import_declaration]
12    import TXLargs [repeat stringlit]
13    deconstruct * TXLargs
14      "−feature" f_name [stringlit] more_options [repeat stringlit]
15    construct f_name_id [id]
16      _ [+ f_name]
17    deconstruct block_imp
18      '#feature f_name_imp [id]
19      '{
20        extracted_imp [repeat block_or_normal_import_declaration]
21      '}
22    by
23      extracted_imp [erase_if_different f_name_imp f_name_id]
24      [. rest]
25  end rule
26
27  function erase_if_different f_name_imp [id] f_name [id]
28    replace * [repeat block_or_normal_import_declaration]
29      imp [repeat block_or_normal_import_declaration]
30    where not
31      f_name_imp [= f_name]
32    by
33      %empty
34  end function
```

I.h Erase if empty

The next file contains the rules that are used to remove empty classes from the file and after removing them to check if the file is empty and remove it if necessary.

```
1   function erase_if_empty
2     replace [program]
3       p [program]
4     by
5       p [change_marks]
6         [erase_if_other_feature]
7         [check_if_empty]
8   end function
9
10  function change_marks
11    replace * [repeat type_declaration]
```

```
12      dcl [repeat type_declaration]
13    by
14      _ [change_class_mark each dcl]
15  end function
16
17  function change_class_mark dcl [type_declaration]
18    replace [repeat type_declaration]
19      t [repeat type_declaration]
20    construct new_dcl [type_declaration]
21      dcl [change_class_mark2]
22    by
23      t [. new_dcl]
24  end function
25
26  function change_class_mark2
27    replace [type_declaration]
28      mark [feature_mark]
29      dcl [normal_type_declaration]
30    import TXLargs [repeat stringlit]
31    deconstruct * TXLargs
32      "-feature" f_name [stringlit] more_options [repeat stringlit]
33    construct f_name_id [id]
34      _ [+ f_name]
35    construct new_mark [feature_mark]
36      '#feature f_name_id
37    where
38      dcl [?has_class_body_declaration]
39    by
40      new_mark
41      dcl
42  end function
43
44  function has_class_body_declaration
45    replace * [class_body_declaration]
46      cbd [class_body_declaration]
47    by
48      cbd
49  end function
50
51  rule erase_if_other_feature
52    replace * [repeat type_declaration]
53      dcl [type_declaration]
54      rest [repeat type_declaration]
55    import TXLargs [repeat stringlit]
56    deconstruct * TXLargs
57      "-feature" f_name [stringlit] more_options [repeat stringlit]
58    construct f_name_id [id]
59      _ [+ f_name]
60    deconstruct * [feature_mark] dcl
61      '#feature f_name_dcl [id]
62    where not
63      f_name_id [= f_name_dcl]
64    by
65      rest
66  end rule
67
68  function check_if_empty
69    replace [program]
70      p [program]
71    where not
72      p [?has_class_body_declaration] [?has_this_feature_mark]
73    by
74      '#empty
75  end function
76
77  function has_this_feature_mark
78    replace * [feature_mark]
79      '#feature f_name_this [id]
80    import TXLargs [repeat stringlit]
81    deconstruct * TXLargs
```

```
82      "-feature" f_name [stringlit] more_options [repeat stringlit]
83    construct f_name_id [id]
84      _ [+ f_name]
85    where
86      f_name_id [= f_name_this]
87    by
88      '#feature f_name_this
89  end function
```

I.i Grammar overrides

This section contains the grammar overrides to the default Java grammar in order to be able to mark the code and apply the transformation rules.

```
1   compounds
2     '#feature
3     '#del_modif
4     '#hook
5     '#empty
6     ...
7   end compounds
8
9   redefine program
10      '#empty
11    | ...
12  end redefine
13
14  redefine class_body_declaration
15      [marked_class_body_declaration]
16    | [normal_class_body_declaration]
17  end redefine
18
19  define marked_class_body_declaration
20    [repeat extraction_mark+]
21    [normal_class_body_declaration]
22  end define
23
24  define normal_class_body_declaration
25        [empty_declaration]
26      |   [member_declaration]
27      |   [instance_initializer]
28      |   [static_initializer]
29      |   [field_declaration]
30  end define
31
32  redefine statement
33    [marked_block]
34    | ...
35  end define
36
37  define marked_block
38    [repeat extraction_mark+] [NL]
39    [block] [NL]
40  end define
41
42  define extraction_mark
43    [feature_mark]
44    | [delete_modifier_mark]
45    | [hook_method_mark]
46  end define
47
48  define feature_mark
49    [NL] '#feature [feature_name]
50  end define
51
52  define feature_name
```

```
53    [id]
54  end define
55
56  define delete_modifier_mark
57    [NL] '#del_modif [list modifier+] [opt applied_mark]
58  end define
59
60  define applied_mark
61    'applied
62  end define
63
64  define hook_method_mark
65    '#hook [feature_name] ', [feature_name] [hook_declaration] [hook_call]
66  end define
67
68  define hook_declaration
69    [repeat modifier] [opt generic_parameter] [type_specifier] [method_declarator] [opt throws]
70  end define
71
72  define hook_call
73    [declaration_or_statement]
74  end define
75
76  redefine type_declaration
77      [marked_type_declaration]
78    | [normal_type_declaration]
79  end redefine
80
81  define marked_type_declaration
82    [feature_mark]
83    [normal_type_declaration]
84  end define
85
86  define normal_type_declaration
87        [class_declaration]       [NL][NL]
88      |   [interface_declaration]  [NL][NL]
89      |   [enum_declaration]        [NL]
90  end define
91
92  redefine package_declaration
93    [opt package_header]
94      [repeat block_or_normal_import_declaration]
95      [repeat type_declaration]
96  end redefine
97
98  define block_or_normal_import_declaration
99      [block_import_declaration]
100   | [import_declaration]
101 end define
102
103 define block_import_declaration
104   [feature_mark] [NL]
105   '{ [NL]
106     [IN] [repeat block_or_normal_import_declaration] [EX]
107   '} [NL]
108 end define
```

References

[1] Argouml. http://argouml.tigris.org/.

[2] Asf+sdf meta environment. http://www.meta-environment.org/.

[3] Dms software reengineering toolkit. http://www.semanticdesigns.com/Products/DMS/DMSToolki

[4] Javapp. http://www.slashdev.ca/javapp/.

[5] Software product lines. http://www.softwareproductlines.com.

[6] Stratego/xt. http://strategoxt.org/.

[7] TXL, 2010. http://www.txl.ca/.

[8] Velocity Template Engine, 2010. http://http://velocity.apache.org/.

[9] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, pages 221–231. IEEE, 2009.

[10] AspectJ. http://eclipse.org/aspectj/.

[11] D. Batory. AHEAD Tool Suite, 2010. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[12] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[13] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.

[14] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[15] James R. Cordy. Excerpts from the txl cookbook. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 6491 of *Lecture Notes in Computer Science*, pages 27–91. Springer, 2009.

[16] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.

[17] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM (CACM)*, 44(10):29–32, 2001.

[18] IRILL. Coccinelle. http://coccinelle.lip6.fr/.

[19] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[21] Christian Kner. Cide: Decomposing legacy applications into features. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 149–150. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[22] Hongzhi Liang and Jürgen Dingel. A practical evaluation of using txl for model transformation. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2008.

[23] Jia Liu, Don S. Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 112–121. ACM, 2006.

[24] Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In Jan Bosch, editor, *GCSE*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[25] Roberto E. Lopez-Herrejon, Leticia Montalvillo-Mendizabal, and Alexander Egyed. From requirements to features: An exploratory study of feature-oriented refactoring. In *SPLC*, pages 181–190. IEEE, 2011.

[26] Eclipse Project. Eclipse ide. http://www.eclipse.org/.

[27] S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *ICSE*, 2007.