Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

School of Engineering and Sciences

Semi-Automatic Program Partitioning

A dissertation presented by

Priscila Angulo López

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In

Information Technologies and Communications

Major in Computer Science

Monterrey, Nuevo León, December 5th, 2017

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey
School of Engineering and Sciences

The committee members, hereby, certify that have read the dissertation presented by **Priscila Angulo López** and that it is fully adequate in scope and quality as a partial requirement for the degree of **Doctor of Philosophy in Information Technologies and Communications**, with a major in **Computer Science**.

Committee members:

Don Batory

Dr. Don Batory
University of Texas at Austin
Advisor

[illegible]

Dr. Guillermo Jiménez Pérez
Tecnológico de Monterrey
Advisor

[illegible]

Dr. Ramón Brena Pinero
Tecnológico de Monterrey
Advisor

[illegible]

Dr. Juan Carlos Lavariega Jarquín
Tecnológico de Monterrey
Committee Member

[illegible]

Dr. Leonardo Garrido Luna
Tecnológico de Monterrey
Committee Member

[illegible]

Dr. Juan Arturo Nolazco Flores
Tecnológico de Monterrey
Committee Member

Tecnológico de Monterrey
05 DIC 2017
Dirección Nacional de Posgrado
Dr. Rubén Morales Menéndez

[illegible]

Dr. Rubén Morales Menéndez
Dean of Graduate Studies
School of Engineering and Sciences

Monterrey, Nuevo León, December 5th, 2017

Declaration of Authorship

I, **Priscila Angulo López**, declare that this dissertation titled, **Semi-Automatic Program Partitioning** and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this dissertation has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the dissertation is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

[illegible]

Priscila Angulo López
Monterrey, Nuevo León, December 5th, 2017

©2017 by Priscila Angulo López
All rights reserved

Dedication

A mi amada madre.

Acknowledgements

This was a long endeavor, accompanied by added trials, as life usually goes. However, the best part is that it allowed me to be the recipient of exceptional encouragement, support and love from a large number of generous and kind persons. I would like to mention them in chronological order.

Professor Guillermo Jiménez, who has been my supporter since before this endeavor and, planted this Ph. D. idea in my mind, thank you for opening my eyes to a world of opportunities that I never thought were within my reach. **Luis**, thank you for pushing me to be brave and assertive. **Mom and Dad**, thank you for your unconditional love, for being my pillars, and for helping me to stand up after every fall, I love you both. **Rosy Sandoval**, thank you for your long and loyal friendship. You are my family, my home. Thank you for your unconditional love. **Professor Lorena Gómez**, thank you for being my mentor and my friend. You further set me in the right direction. Thank you **Manuel Rodríguez**, my dear friend. You are my example of endurance and resilience. **Pablo Mazzucchi and Mónica Muñoz**, thank you for every piece of advice, and for extending your home's warmth to me. **Marcel Váldez**, you are a great example of quality and excellence. Thank you for teaching me how to be thorough. **Rachel Lloyd**, now Rachel Ahn, my best friend, thank you for your kindness, generosity and constant love. **Carl W. Handlin**, your friendship and support arrived at the most critical moment, thank you for being my friend and the best teammate. **Néstor Vázquez**, I admire you and deeply appreciate every perspective-giving conversation we have. **José Luis Leal**, thank you for listening to me and giving me strength every time I needed it. **Eduardo González**, thank you for your support and for always being full of empathy. **Seth Beinhart**, thank you for your constant support and endless encouragement. **Mariel Mariscal**, my dear cousin, confidant, my shoulder to lean on: You are irreplaceable. **Fernando Reyes**, thank you for your encouragement and your positivism. **Don and Karen Batory**, my Austin *adoptive* parents, thank you for embracing me, comforting me, bearing with me, and for making me the object of your warmth and kindness. **Mireya Almaraz**, my little sister I never had, thank you for your example of perseverance and your loving support. **Ana Rodríguez**, thank you my friend for your generosity and our reassuring conversations. **Fernando Sánchez**, **Pedro Pérez** and **Tiago Damasceno**, thank you my friends for your support and company. **Kennedy C. Brown**, thank you for your support, your friendship, and being my affirmative company during stressful times. **Vandana Gunupudi**, thank you my friend for every time you gave me reassurance and inspired strength in me. **Deven Desai**, thank you for your friendship and constant words full of empathy and consideration.

I regret that I can only list each of your names once, as you have each helped me not once, but a multitude of times. Thank you.

Abstract

Partitioning a program into features is a fundamental problem in both *Software Product Line* (SPL) and *object-oriented framework* research.

An SPL is a family of related programs that are created and differentiated by *features* – increments in program functionality. An extractive way to create an SPL is to refactor a legacy program into a stack of feature modules or layers. We developed B^t_tF, a tool that infers the feature membership of all package, class, and member declarations of a Java program from facts given by a program expert. B^t_tF recognizes *hooks* to be methods that defy classical layering in that they reference declarations of layers/feature modules *that are added in the future*. The information collected by B^t_tF is used to feature-annotate the program, giving it a feature-based design. Doing so enables optional or unnecessary features to be removed from the program's codebase to produce subprograms, the beginnings of an SPL. Case studies validate B^t_tF, the best result yielded by our experiments show that B^t_tF is capable of inferring the assignment of over 80% of a program's declarations, this means that less than 20% of the assignments were provided as facts by an expert.

An object-oriented framework is a two-tiered structure: a *framework* (consisting of many abstract classes) that encode common and shared behavior in a domain of applications, and a *plug-in* (having a concrete class for each framework abstract class) to complete an application. Refactoring a legacy application into a framework and plug-in is a common activity. Surprisingly, tools to perform this task are absent, even though frameworks have been in use for 25 years. After all this time, it remains a tedious, error-prone, and manual task. Two tools are needed: one that determines how classes of a legacy application and their members are partitioned across a framework and plug-in, and which methods are hooks. A second tool takes this partitioning information as input and runs a refactoring script that invokes refactorings programmatically to transform the legacy application into a framework and a plug-in. `R3` is a new refactoring engine for Java that supports the writing of transformation scripts [51]. We present the integration of B^t_tF and `R3` to cover this need, and evaluate this integration by effectively transforming a set of six Java programs into a framework and plug-in structure.

Furthermore, we use B^t_tF as a tool to identify features in a non-object-oriented environment: *Multi-Agent Systems* (MAS). This active research area has started to see some efforts to integrate SPL concepts and technologies, however, identifying features in existing MAS was a problem not being addressed. B^t_tF was successfully used to identify features on six MAS simulators developed on `GAMA platform`.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

A *Software Product Line* (SPL) is a design for a family of related programs that are differentiated by *features* — stereotypical increments in program functionality [1]. Every program of an SPL is theoretically a stack of layers or (equivalently) feature modules [12, 94, 92].

SPLs offer significant improvements in cost, time-to-market, and quality in customized program production [17]. However, adopting SPL technology is a considerable challenge, it requires converting legacy programs into an SPL, a notoriously difficult task [47, 98, 23, 56, 29].

Why is this task hard? Consider what the task implies:

1. Identify a set of features in a program, then
2. Partition the original codebase by indicating which feature introduced each declaration (or more microscopically, which lines of code); the code of a feature is the contents of its feature module,
3. Determine the correct order in which these features were incrementally added to the program, and which feature depends on which other features,
4. Verify that you did this process correctly by removing one (say an optional feature) and still have the resulting program correctly compile.

Now consider, where to start to accomplish this task? Related research suggests searching for keywords related to a feature in comments and source code, or perform pattern matching on naming conventions [23, 29, 47, 56, 98]. This could be useful for step 1, but it is methodologically incomplete, and is far from guaranteeing the achievement of our final goal: a *correct feature partitioning of the program.*

Prior work does not exploit the structure of feature modules to extract an SPL. Nor do they leverage 'feature-`private`' modifiers of packages, classes, and members to limit their visibility only to that module, as opposed to 'feature-`public`' declarations that are visible to all feature modules. Experience in building Java SPLs has long hinted that 'feature-`private`' declarations could help identify the contents of a feature module as any declaration that references a 'feature-`private`' element must also belong to that module.

We developed *Back to the Future* (B^t_tF), a new tool to extract an SPL from a legacy program.[1] Unlike most prior work, B^t_tF does *not* use feature location, concern location heuristics, text analysis or machine learning to extract an SPL. Most prior work based their approaches analyzing source *text*, and cannot offer strong guarantees that their results are consistently successful. Instead, B^t_tF leverages a structural theory of SPLs whose programs are composed from feature modules. This required a close and precise integration of previously loosely-connected ideas: truth maintenance systems [38], classical layering [28], mixin-layers [6, 92, 94], hook methods in OO frameworks [79], and 'feature-`public`' and 'feature-`private`' modifiers [3] that would adorn declarations in feature modules if such modules were supported by Java. B^t_tF is *semi-automated*: given a Java legacy program $\mathbb{P}$ and a stack of features, it relies on a program expert to supply facts about $\mathbb{P}$'s declarations to infer a feature partition of all $\mathbb{P}$ declarations.

Our work is based on a theory of layering [93, 94]. Leveraging layer concepts guarantees the type correctness of a partition (feature). A layer depends on previously introduced layers to compile. We treat features as layers, we stack them in the order they should be introduced, guaranteeing the type correctness of the partition. This was not our first approach, and definitely not an approach taken by related work. Originally, we tried to build upon existing work, but their lack of guarantees made us develop our own approach. We first thought of using SAT solver concepts [1], however we promptly realized that it was insufficient. The right approach became clear when we started representing a program declaration dependencies as a graph. It was then when layers became an obvious partitioning approach. The rest fell in place in an almost organic way.

B^t_tF is integrated with `R4`, an Eclipse plug-in, that supports the refactoring and creation of feature-annotated Java SPL codebases [50]. `R4` plays a similar role to that of the *C-PreProcessor* (CPP) used in C-based SPLs, except it uses Java annotations. But `R4` also supports the refactoring of such codebases, where only limited support exists today for C+CPP SPL codebases [55]. B^t_tF takes the feature-assignments it is given and infers and feature-annotates all $\mathbb{P}$ declarations to create a Java SPL that `R4` can process.

A task that is similar to converting a legacy program into an SPL is converting a program into a framework and plug-in. A *framework* is a classical design in reusable *Object-oriented* (OO) software [21, 36, 45, 67, 79]. It standardizes the encoding of the classes, relationships, and functionality that are shared by programs in an application domain. Specific applications extend/complete the framework by a *plug-in* which adds the distinctive functionality of that application. Frameworks abound in graphics [101], middleware [87], web application frameworks [88], IDEs [33, 64], and, of course, software product lines [9].

A common way to create a framework is to refactor an existing legacy application into two parts: a reusable framework and an application-specific plug-in. The basic ideas for doing so have been known for 25 years. Yet after all this time, it remains a tedious, error-prone, and *manual* task. For example, a legacy class might (1) belong to the framework, (2) belong to the plug-in, or (3) have its members partitioned so that some belong to the framework and others to the plug-in. This activity of classifying the contents of a class must be made

[1] Mainly executed as an Eclipse plug-in.

repetitively for every class in the legacy application. Even for legacy applications with a few classes, without tool support it is easy for programmers to get lost or make mistakes. We know of no widely available tool(s) that help.

A second task takes the classification output of the first task and refactors each legacy class, either moving it into a framework or plug-in, or to distribute its members across both. The difficulty here is that refactoring engines of today's IDEs allow programmers to manually perform ***one*** refactoring at a time. When tens or hundreds of refactorings are needed, once again programmers can easily get lost or make mistakes. What is missing is the ability to script refactorings (*ie* to programmatically invoke refactorings repeatedly) which would make this second task largely automatic.

We use B^t_tF as a support tool to partition a Java legacy program into framework and plug-in. This information is then input to a refactoring script that automatically transforms the legacy application into a framework and plug-in. This script is executed by a new refactoring engine R3 for Eclipse that supports user-programmable refactoring scripts as simple Java methods [51]. To give readers an appreciation, most design patterns (in the Gang-of-Four text [108]) can be encoded as an R3 Java method of ~ 25 LOC. The B^t_tF legacy program to framework + plugin refactoring script is ~ 300 LOC, the largest script written to date.

Furthermore, the need to identify features is not only relevant for OO programs. It is also important in other software development scenarios, for example, *Multi-Agent Systems* (MAS) development. Among the research topics to advance MAS engineering, is the effort to combine MAS and SPL approaches: *Multi-Agent Systems Product Lines* (MAS-PL) [75]. Most of the research done in MAS-PL is in methodological aspects, and there is no tool to identify features in a MAS. We use B^t_tF also in this scenario.

The novel contributions of this research are:

- A careful and precise integration of the concepts of Java packages, classical layers, mixin-layers, hook methods, feature-modifiers, and truth maintenance systems;
- The *Containment Reference Graph* (CRG) as the key data structure for SPL extraction analysis;
- Rules to infer and maintain a consistent database of facts (feature-declaration assignments);
- Case studies that demonstrate the utility of B^t_tF for SPL partitioning, framework and plug-in refactoring, and MAS features identification; and
- An explanation of how B^t_tF goes beyond prior work on feature-localization and extracting SPLs.

We begin by with an overview of relevant prior work.

Chapter 2

Background

2.1 Layers and Features

Layering is a fundamental design technique to control and manage software complexity [12, 28]. Layers exist in every non-trivial program explicitly or implicitly [43]. Figure 2.1 illustrates classical layering. Each layer is a blue-colored horizontal rectangle that contains a set of classes. A package is a vertical black rectangle. In classical layering, there is no difference between layers and packages; *eg* layer Lη equals package Pη. The lowest layer or *base* is `L1` and the highest is `L3`. We write this stack of layers as `L1`$\times$`L2`$\times$`L3`, where $\times$ is the layer stacking operation.

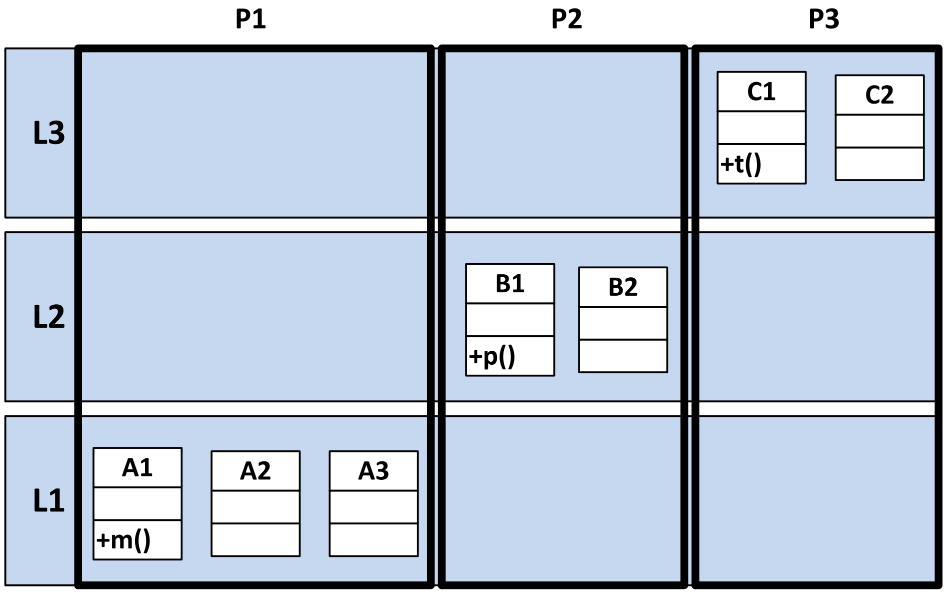


Figure 2.1: Classical Software Layers.

Suppose all declarations are either `private` or `public`; there is no package-`private` (or null) modifiers as in Java. Then:

- L1 has three classes A1–A3. Method A1.m can reference private members in class A1 and public members in L1.
- L2 has two classes B1 and B2. Method B1.p can reference private members in B1 and public members in L1 × L2.
- L3 has two classes C1 and C2. Method C1.t can reference private members in C1, public members in L1 × L2 × L3.[1]

2.1.1 Object Oriented Layering

OO layering has a more sophisticated structure [92, 94]. The essential idea is that each subsequent layer can not only add new classes, but also extend, via subclassing, any public class of a layer beneath it, this can be observed in Figure 2.2.

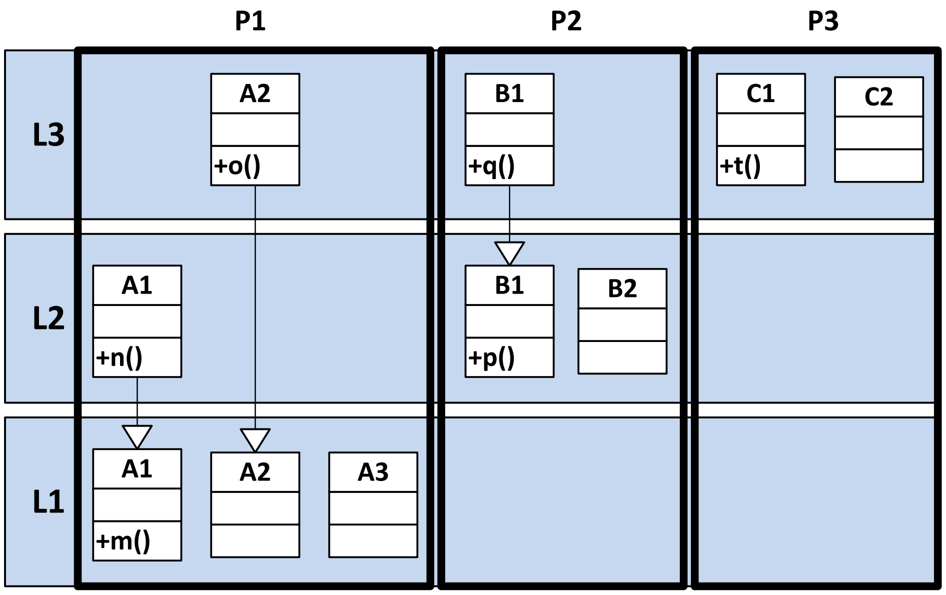


Figure 2.2: Illustrates OO layering.

- L1 is indistinguishable from its classical counterpart.
- L2 introduces classes B1 and B2 as before. But now it also extends class L1.A1 with method n. An extension, like any subclass, can add new fields and methods to its parent class, and override existing methods. As in classical layering, methods of L2 can reference public members in L1 × L2.
- L3 again introduces classes C1 and C2. It also extends class L1.A2 with method o, and extends class L2.B1 with method q. Methods of L3 can reference public members in L1 × L2 × L3.[2]

[1] Early papers on classical layering restricted the access of L_i members to public members of the layer immediately below it, namely layer L_{i-1} [28]. This is too strict for practical layering [12].

[2] By historical accident, inheritance or subclassing hierarchies are normally drawn with the base class at

2.1.2 Legacy Programs Telescope OO Layers

Habermann et al argued that layers exist in every application; most layers are implicit, not explicit [43]. This means that instead of having explicit OO layers as in Figure 2.2, programmers telescope or collapse sub-classing chains into a single class, removing all traces of explicit layering. The result is a monolithic layer or program $\mathbb{P}$ of Figure 2.3, which to us is the OO structure of a *legacy program*.

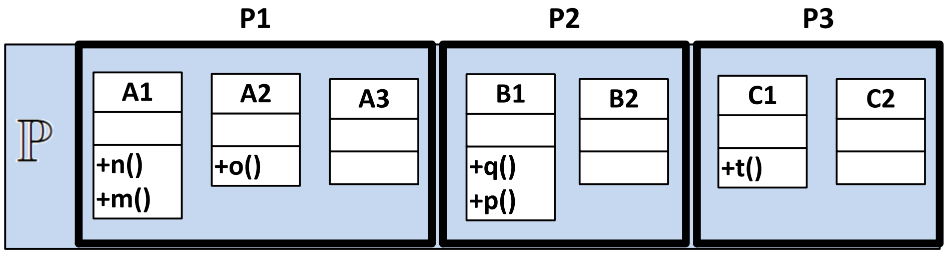


Figure 2.3: A Legacy Program Telescoped from Figure 2.2.

This is our starting point. We are given a telescoped legacy program $\mathbb{P}$ that is a collection of packages. Using the interactive analysis of `BttF`, a programmer tells us that $\mathbb{P}$ has the layer stack `L1`×`L2`×`L3`, and from facts (s)he gives us, we infer the contents of these layers.

2.1.3 OO Layers, Packages, and Features

OO layers are collections of classes and so too are packages. The difference: an OO layer cross-cuts packages. The dark outlines in Figure 2.2 show the contents of packages `P1`−`P3`, and their telescoped counterparts in Figure 2.3.

OO layers correspond to features in SPLs [5, 6, 2]. Henceforth we refer to OO layers as features.

2.1.4 Feature Modifiers

Modifiers `private`, `<null>` (package-`private`), and `public` adorn every declaration in a Java package. They tell Java the visibility of a declaration is restricted to a single class, to a package, or to the world.

Features also have modifiers [3]. `fpublic` says a declaration is exported or is visible to other features of a program. In contrast, `fprivate` says a declaration is visible to (or can be referenced by) *only* the feature which defined it.

the top of a hierarchy and the most specialized subclasses at the bottom. This is in contravention to layer hierarchies. We retained classical layer ordering in Figure 2.2, requiring the base class of a subclassing chain to be at the bottom, not at the top. This is why the inheritance chains of Figure 2.2 are inverted.

Java modifiers are independent of feature modifiers, although this is not immediately apparent. (Apel et al independently reached a similar conclusion [3]). Let's contrast Java `private` and feature `fprivate`. Figure 2.4 depicts a Java package with the feature stack `F1`×`F2`×`F3`. Feature `F1` introduces base versions of classes `J`, `K`, `L`; features `F2` and `F3` extend these classes. Member `J.a` is introduced by feature `F1`. If `J.a` is `fprivate`, it is *horizontally* visible to all classes of `F1` and is otherwise *invisible* to (or not referenceable by) other features. If `J.a` is `private`, it is *vertically* visible to all later extensions of class `J`, but to no other classes. Having both `private` and `fprivate` modifiers restricts `J.a`'s visibility to their overlap, namely the declarations that `F1` introduced in class `J` (bottom left class of Figure 2.4).

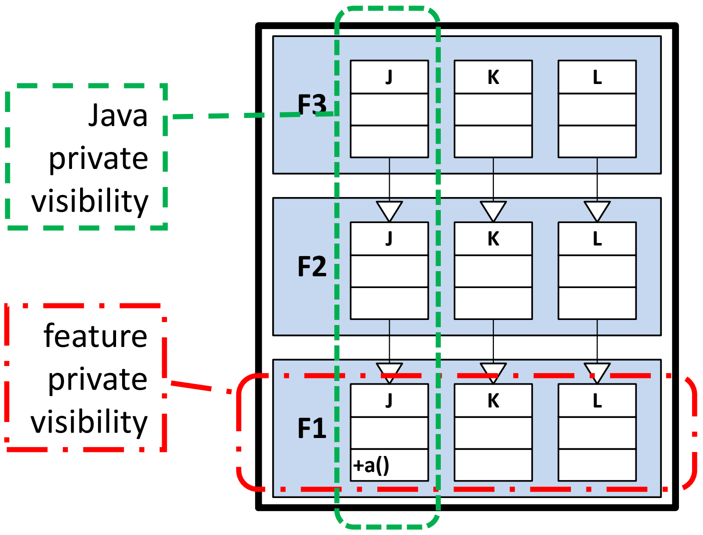


Figure 2.4: Differences between `private` and `fprivate`.

Now contrast Java `public` and feature `fprivate` (Figure 2.5). If `J.a` is `public` then `J.a` is visible to the world. But if `J.a` is also `fprivate`, *only the code of feature* `F1` *can reference* `J.a`*; it is invisible to (not referenceable by) other features.*

The remaining combinations also yield surprises. If member `J.a` was `fpublic` in Figure 2.4, its visibility would be indistinguishable from that of Java package-`private`. But if `J.a` was `fpublic` in Figure 2.5, `J.a` would be visible only to packages `P1`–`P3`; there is no corresponding mechanism in Java that would restrict its visibility only to those packages. For the above reasons, we concluded that Java and feature modifiers are independent.

Obviously, feature modifiers are not present in a Java legacy program. Because we can treat feature modifiers independently of Java modifiers, this simplifies our task. *We do not alter the Java modifiers in feature-refactoring* $\mathbb{P}$. To feature-refactor program $\mathbb{P}$ into a stack of features, we must recover for each declaration of $\mathbb{P}$ (packages, classes, initializers, constructors, and members):

- the feature to which it belongs and
- whether or not it is `fprivate`.

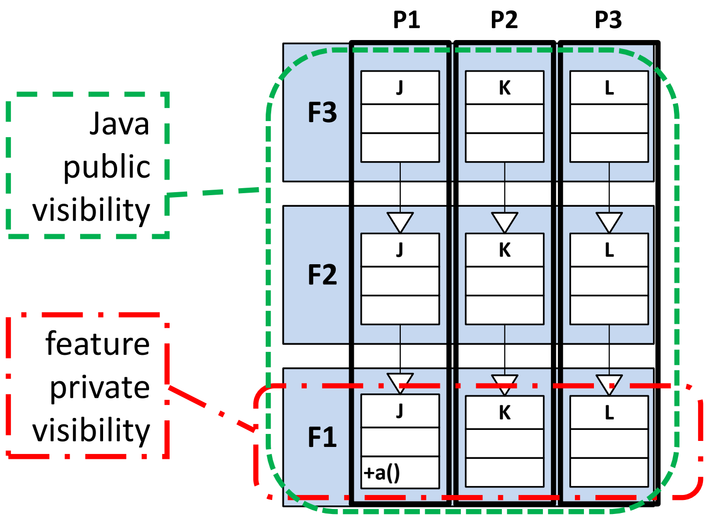


Figure 2.5: Differences between `public` and `fprivate`.

So if ℙ is the program of Figure 2.3 and its feature stack is L1×L2×L3, our task is to reconstruct the design of Figure 2.2.

2.2 The Challenge and Relevance of SPLs

A *Software Product Line* (SPL) is a design for a family of related programs [1]. Each program is composed from a unique stack of feature modules or layers, each of which is a stereotypical increment in program functionality [12, 92, 94]. SPLs offer significant improvements [17], [78, pp. 9-11]. Namely, they:

- improve the software development process,
- reduce cost and time to market,
- increase software quality attributes like reliability, usability and maintainability,
- provide increased ability of application evolution, and
- reduce application complexity, and increase manageability.

Two fundamental barriers limit SPL success. First, there are astonishingly few `public` Java SPLs. Experimental studies on real SPLs are critical to advance SPL technology [63, 72]. Second, adopting SPL technology requires *converting* legacy programs into an SPL, a notoriously difficult task [23, 29, 47, 56, 86, 98].

Although many tools can search code for keywords to locate features or concerns, *ie* positions in source code that *might* implement a desired functionality [29, 86], few tools extract a Java SPL from a legacy program [48, 76, 99].

Prior work does not exploit the structure of feature modules to extract an SPL. Nor do they leverage the concept of feature-`private` modifiers of packages, classes, and members to limit their visibility only to that module, as opposed to feature-`public` declarations that are visible to all modules. Our experience in building Java SPLs has long hinted that feature-`private` declarations could help identify the contents of a feature module as any declaration that references a feature-`private` element must also belong to that module.

An SPL can be created in three ways [1]:

1. A *proactive* way develops a product line from scratch;
2. A *reactive* way extends a small product line incrementally with new features, broadening the product line's scope; and
3. An *extractive* way starts with a legacy program (sometimes several) and feature-refactors it into an SPL, at which point a reactive approach takes over.

B$^{t}_{t}$F (Back to the Future) is a new tool to extract an SPL from a legacy program. Unlike most work, B$^{t}_{t}$F does *not* use feature location or concern location heuristics to extract an SPL. Instead, B$^{t}_{t}$F leverages a well-known structural theory of SPLs whose programs are composed from feature modules. This required a close and precise integration of previously loosely-connected ideas: truth maintenance systems [38], classical layering [28], mixin-layers [6, 92, 94], hook methods in OO frameworks [79], and feature-`public` and feature-`private` modifiers [3] that would adorn declarations in feature modules if such modules were supported by Java. B$^{t}_{t}$F is semi-automated: given a Java legacy program $\mathbb{P}$ and a stack of features, it relies on a program expert to supply facts about $\mathbb{P}$ to infer a feature partition of all $\mathbb{P}$ declarations.

2.3 Framework and Plug-in Concepts

Frameworks are a standard design for OO extensibility [46, 37, 15]. A *framework* consists of one or more `abstract` classes that define a common functionality in an application domain. It might also have zero or more additional `private` concrete classes that are required for this functionality. Each `abstract` class has one or more *hook methods* – `abstract` methods whose implementation must be specified by a plug-in.

A *plug-in* or extension, is a set of concrete subclasses, one for each framework `abstract` class, that provides additional functionality to complete the framework. A plug-in's concrete classes implement the framework's hook methods. Like the framework itself, a plug-in can have additional `private` concrete classes that complete the plug-in's functionality [57, 15].

Figure 2.6a shows a simple framework $\mathscr{F}$ consisting of two `abstract` classes (`A1` and `A2`) and one concrete class (`X1`). Every framework also has an *`abstract` factory* class, `AF`, which provides an `abstract` factory method for each `abstract` and/or `public` class in the framework. (Note: blue classes and methods are `abstract` in our figures).

Figure 2.6b shows a plug-in $\mathscr{P}$. There is a concrete subclass `C` for each framework `abstract` class `A` and a concrete method `C.h()` for each `abstract` hook method `A.h()`. Two such classes

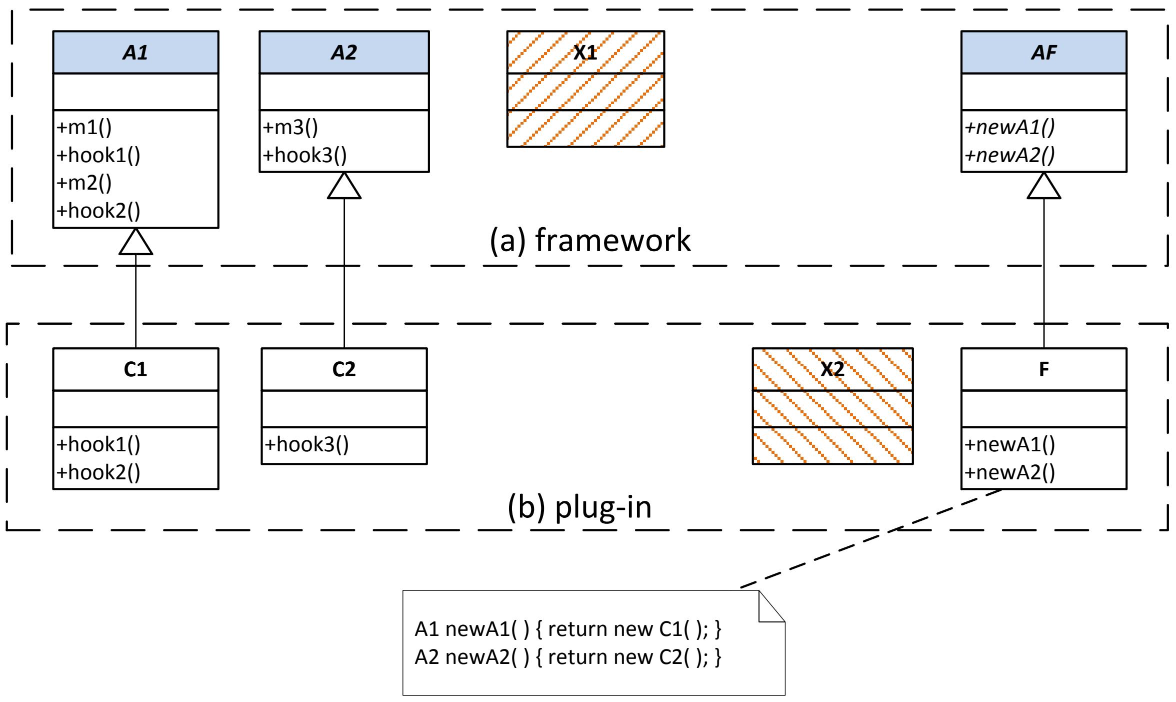


Figure 2.6: Basic Framework and Plug-in Organization.

are shown in Figure 2.6b, `C1` and `C2`. An additional concrete class specific to the plug-in (`X2`) is also present. Plug-in $\mathscr{P}$ is completed with a concrete factory class `F` that has a factory method for each `public` class constructor in the framework + plug-in combination. Note that concrete factory method `newA1()` returns a (plug-in) `C1` object; method `newA2()` returns a (plug-in) `C2` object.

2.3.1 Partitioning $\mathscr{L}$ into $\mathscr{F}$ + $\mathscr{P}$

Legacy application $\mathscr{L}$ is mapped to a framework $\mathscr{F}$ and plug-in $\mathscr{P}$ in three steps. First, each class of $\mathscr{L}$ is either (1) moved to $\mathscr{F}$ or (2) moved to $\mathscr{P}$ or (3) its membership is split between $\mathscr{F}$ and $\mathscr{P}$. Steps (1) and (2) are implemented by a move-class refactoring. Figure 2.7 shows the typical steps of option (3). Starting with class `C1`, an expert determines that methods `m1()` and `m2()` have both framework content and plug-in content. So these methods are split (lifted) into their generic part (whose names remain `m1` and `m2`) and hook methods `hook1()` and `hook2()` that are respectively called from `m1` and `m2` and that contain plug-in contents.

Second, the revised class `C1` is partitioned into an `abstract` class `A1`, which contains all framework-classified members, while class `C1 extends A1` retains the concrete members that are specific to the plug-in. This transformation (of moving or splitting) is applied to all classes in $\mathscr{L}$.

Third, the framework $\mathscr{F}$ is given an `abstract` factory class `AF` (containing abstract factory methods) and the plug-in $\mathscr{P}$ is given the corresponding concrete factory subclass `CF` (con-

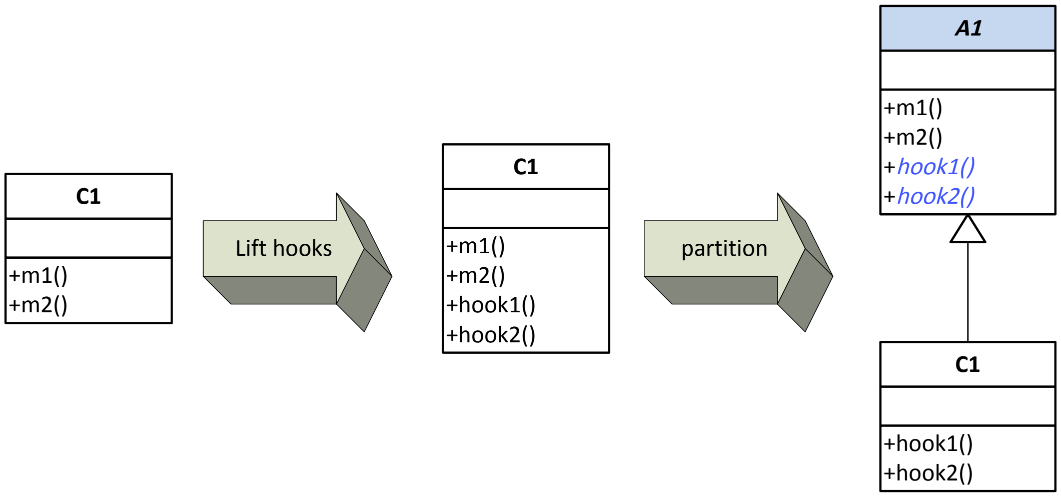


Figure 2.7: Two-Step Splitting of Class `C1`.

taining corresponding concrete factory methods).

This is the essential mapping of $\mathscr{L}\rightarrow\mathscr{F}+\mathscr{P}$. It can be implemented by a R3 refactoring script.

2.3.2 R3 Refactoring Scripts

R3 is a new Java refactoring engine for Eclipse [49, 51].[3] R3 was specifically designed to support refactoring scripts as Java methods to invoke refactorings programmatically. In particular, most classical design patterns can be expressed as 10-25 line methods, which has been the primary use for R3 to date [49, 51]. R3 does not rely on the Eclipse refactoring engine, but rather uses a novel relational-database and pretty-printing technology to map Java programs to refactored Java programs [51].

R3 works in the following way: R3 exposes all packages, classes, methods and fields as R3 objects, similar to that of reflection of Java class files. What distinguishes R3 from reflection are R3 object methods: they are refactorings (*eg* rename, create, move) or methods to find related objects (*eg* return the R3 class object of a given method, return the R3 objects that are members of a given class, return all R3 method objects that have a given type signature).

An R3 script for the `moveAndDelegate()` refactoring is shown in Figure 2.8. The script delegates a method `m` to a class whose name is `cn` in a package whose name is `pn`. Line 3 determines the R3 class object `o` to which method `m` belongs. Line 4 converts name `pn` to a R3 package object `p`. Line 5 converts name `cn` to an R3 class object `c`. Line 7 moves method `m` to class `c` and Line 8 adds a delegate method for `m` to class `o`. Errors in R3 script execution are Java exceptions; as R3 does not alter Eclipse *Abstract Syntax Trees* (ASTs), recovery from failed script executions is fast.

[3] R4 is a Java SPL refactoring engine that is an extension of non-SPL refactoring engine R3.

```
void moveAndDelegate( RMethod m, String cn, String pn )
{
  RClass o = m.getClass();
  RPackage p = Project.findPackage(pn);
  RClass c = p.getClass(cn);

  m.moveToClass(c);
  o.createDelegate(m);
}
```

Figure 2.8: An R3 `MoveAndDelegate` Refactoring.

The following chapters provide full details on the theory behind B^t_tF, its execution environment, and its successful application and evaluation.

Chapter 3

B^t_tF Classification and Inference Rules

3.1 Introduction

Our tool, *Back to the Future* (B^t_tF)[1], allows the semi-automatic partitioning of a program into a sequence of layers or features.[2] B^t_tF is a *truth maintenance system* [38]. It collects and infers facts, guaranteeing they are consistent. B^t_tF obtains from a user a feature model for partitioning a program and feature + feature modifier assignments to program declarations.

B^t_tF's targeted applications are:

1. Partitioning of a program to create an elementary SPL. We say 'elementary' because the features in the resulting product line are initially all required. However, after this, a user may use other tools to manage and generalize the resulting product line, and derive other products. And

2. Partitioning of a program into *framework+plugin*, the *framework* part is the base layer, the *plugin* part is the second layer with *optional* functionality.

Both applications are very similar, however their outputs are quite different. Details are described later.

The following sections explain B^t_tF partitioning concepts, bounds calculations, and inferences.

[1] B^t_tF's name alludes to a behavior that we noticed while working on this project. When partitioning programs into layers, we found that declarations in a layer would reference declarations in future layers, and future layers would reference declarations defined in past layers.

[2] We use the terms layer and feature interchangeably.

3.2 B$^{t}_{t}$F Concepts

3.2.1 Declaration Types

A *declaration* is a code element found in a program's source. B$^{t}_{t}$F considers the following four declaration types:

- **Package**. A package declaration.
- **Class**. This term includes classes, interfaces, enums and annotations.
- **Method**. Includes methods, constructors, static and non-static initializers.
- **Field**. Represents a member variable. The *initializer expression* (IE) of a field is considered part of the field.

3.2.2 Feature Modifiers

A modifier defines a declaration's visibility among features. There are two possible feature modifiers `fpublic` and `fprivate`; `fpublic` says a declaration is referable by declarations in other features of a program. In contrast, `fprivate` says a declaration is *only* referable by declarations in the feature which defined it.

3.2.3 Facts

A *fact* is provided by a user. It indicates both the feature assignment *and* the feature modifier of a declaration. When B$^{t}_{t}$F infers a feature assignment, it does not infer a modifier. This chapter details B$^{t}_{t}$F inferencing.

3.2.4 Containment Reference Graph (CRG)

The core analysis of B$^{t}_{t}$F relies on a *containment reference graph* (CRG). Each node of a CRG represents a program declaration. A *containment edge* $\mathtt{d} \rightarrow \mathtt{c}$ connects a nested declaration `d` to its container declaration `c`, *eg* `d` is a class, interface, or subpackage of package `c`. A *reference edge* $\mathtt{m} \rightarrow \mathtt{d}$ connects `m` (a method or constructor or initializer or field with an IE) to a declaration `d` that it references. In its analysis, B$^{t}_{t}$F does not distinguish containment from reference edges. We write $\mathtt{m} \rightarrow \{\mathtt{d}, \mathtt{c}\}$ as an abbreviation for $\mathtt{m} \rightarrow \mathtt{d}$ and $\mathtt{m} \rightarrow \mathtt{c}$, and $\{\mathtt{x}, \mathtt{y}\} \rightarrow \mathtt{z}$ for $\mathtt{x} \rightarrow \mathtt{z}$ and $\mathtt{x} \rightarrow \mathtt{z}$.

A CRG is obtained by harvesting the *Abstract Syntax Trees* (ASTs) of a program.[3] References that point to external declarations, like `org.junit`, are excluded. Figure 3.1a presents the code of the Java `Complex` class and Figure 3.1b is its CRG with:

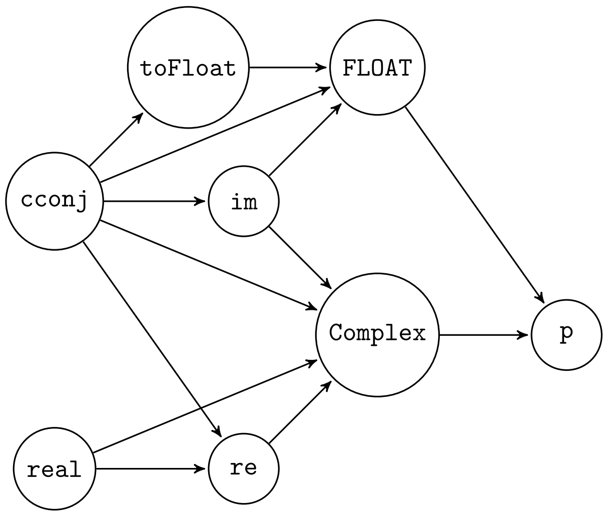
[3] For Java programs ASTs are available in Eclipse. For programs coded in other languages for which compiler support is lacking, other means are needed to obtain a CRG.

- Containment edges: {`FLOAT`, `Complex`} → `p`, `toFloat` → `Float`, {`im`, `re`, `cconj`, `real`} → `Complex`.
- Reference edges: `cconj` → {`toFloat`, `FLOAT`, `im`, `re`} and `real` → `re`.

As readers may observe in Figure 3.1b, fields are not necessarily terminal nodes. Terminals could be any type of declaration that does not reference or contain other declarations.

```java
package p;

class Complex {
  FLOAT im; //FLOAT belongs to p
  float re;

  float cconj() {
    return
        im.toFloat()*im.toFloat()
        + re*re;
  }

  int real() {
    return re;
  }
}
```

(a)



(b)

Figure 3.1: CRG of Java Complex Class

Once a CRG has been obtained, a user must provide a `BttF` feature model to follow for partitioning.

3.2.5 A BttF Feature Model

A *feature model* in `BttF` is provided by a user. Compared to traditional feature models which can be elaborate hierarchies of features with different containment relationships and cross-tree constraints [1], a `BttF` feature model is quite simple – but this is all that is needed for feature-partitioning legacy applications.[4] A program is to be partitioned according to a sequence of features. A `BttF`'s feature model follows this format:

```
P : A B C
```

In this case, `P` is a program to be partitioned into features `A`, `B` and `C`. A feature name is any sequence of letters and numbers, starting with a letter. Feature names can not be repeated. A user provides the names of the features in order as they were layers, starting with the bottom layer or base feature, in this case `A`, and ending with the top layer, in this case `C`.

[4] We call a non-feature oriented program legacy application or legacy program.

A B^t_tF's feature model has only mandatory features, no optional features, and no alternative features[1, 59]. Since we treat features as layers, there are implicit constraints: let F and G be features. If $F < G$, meaning that feature F comes before (or is closer to the base feature) than G, then $G \Rightarrow F$. That is, G requires all features beneath it in a B^t_tF feature model. Cross-tree constraints and more elaborate feature models can be developed from B^t_tF's output using a follow-on Java-based tool for SPL development [50].

A *recursive partitioning* of a feature is its partitioning into a sequence of smaller features. Recursive partitioning has the advantage of divide-and-conquer, which may be helpful to partition large programs. If a feature is recursively partitioned, there will be at least one more line to a B^t_tF feature model that looks like:

```
A : X Y
```

The above means that the base feature, A, is partitioned into a sequence of primitive layers, where the first layer is X and layer Y is atop X. A feature to sub-partition, A above, should exist in one of the previous lines of a B^t_tF feature model. A feature can be sub-partitioned only once. All features (irrespective if they are sub-features or not) have unique names. The sub-features should also be provided in order as they were layers of the containing feature.

3.2.6 Feature Dependency

An edge of a CRG where both connected declarations have been assigned to a feature, provides *feature dependency* information. The feature of the edge's source declaration src(ed) depends on the feature of the edge's destiny declaration dst(ed). Thus, the feature dependency FD of an edge $\mathtt{ed} = \mathtt{src(ed)} \rightarrow \mathtt{dst(ed)}$ is FD(ed):

$$\mathtt{FD(ed)} \quad = \quad \mathtt{F(src(ed))} \rightarrow \mathtt{F(dst(ed))} \qquad (3.1)$$

Feature dependency is related to *Safe Composition* (SC) of an SPL [95]. SC guarantees that all programs in a product line are type safe: there are no references to undefined elements and all programs of the product line compile without error.

Effectively, each edge in a CRG $\mathtt{F(src(ed))} \rightarrow \mathtt{F(dst(ed))}$ is a theorem that Safe Composition has to prove is consistent with the feature model. If it is not, then there will be programs where F(src(ed)) is true and F(dst(ed)) is false, meaning that at least one program of the SPL will not compile.

Note that SC is *not* part of B^t_tF, but of a follow-on tool for SPL development [50]. B^t_tF does not check these rules, as all features that B^t_tF deals with are mandatory, and partitioning is done based on a layered approach, guaranteeing that declarations are introduced in a timely layer- or feature-based manner. Only when features are optional is the full SC analysis needed.

3.2.7 Partitioning Process

B$^{\mathtt{t}}{}_{\mathtt{t}}$F's *main path* to assign features to declarations is the process of visiting nodes of a CRG in a right-to-left breadth-first manner. That is, B$^{\mathtt{t}}{}_{\mathtt{t}}$F traverses the CRG from the leaves to the roots, hence, it first assigns to features those declarations that do not reference other declarations, and then it assigns declarations that point to only leaves, and so on.

B$^{\mathtt{t}}{}_{\mathtt{t}}$F's main path is illustrated in Figure 3.2. This CRG contains declarations `a`, `b`, `c`, `d`, `e`, `f`. The edges are $\mathtt{f} \rightarrow \mathtt{e}$, $\{\mathtt{d}, \mathtt{e}\} \rightarrow \mathtt{c}$, $\{\mathtt{c}, \mathtt{e}\} \rightarrow \mathtt{b}$, and $\mathtt{c} \rightarrow \mathtt{a}$. Declarations `a` and `b` are the leaf nodes, they are assigned first, then `c` and `e`, and finally, the roots `d` and `f`.

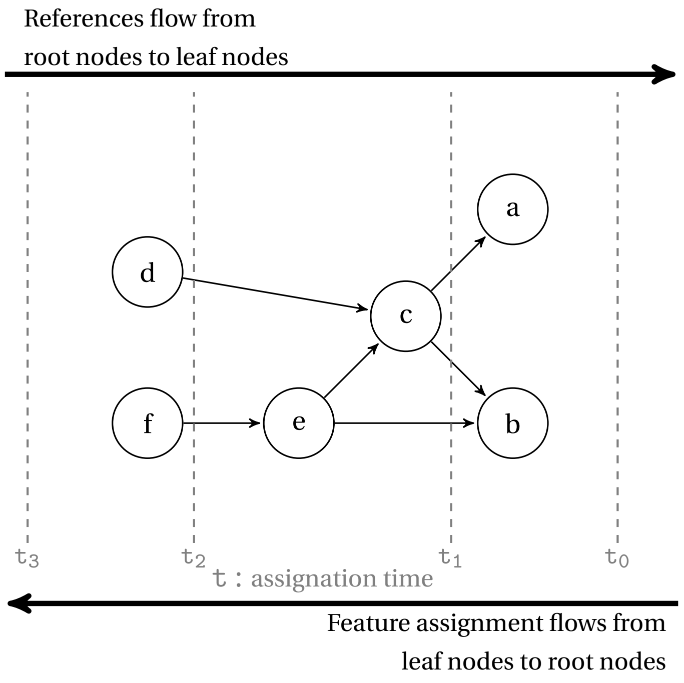


Figure 3.2: B$^{\mathtt{t}}{}_{\mathtt{t}}$F Main Path

The reason for B$^{\mathtt{t}}{}_{\mathtt{t}}$F's main path is historical. Initially, we thought that assigning features to declarations via the main path would be sufficient to feature-partition an application. As it turns out, programmers found it convenient to supply a set of feature and modifier assignments to declarations that are known to them before running B$^{\mathtt{t}}{}_{\mathtt{t}}$F. How B$^{\mathtt{t}}{}_{\mathtt{t}}$F handles this input is covered in Chapter 4.

3.2.8 Notation, Predicates, and Rules

In this and the following sections, we use Greek letters $\alpha \ldots \omega$ to denote features, and their alphabetical linear order $\alpha < \beta < \ldots < \omega$ to denote the order in which they are composed.

$$\alpha \quad \texttt{is the earliest or base feature} \tag{3.2}$$

$$\omega \quad \texttt{is the latest, last or top feature} \tag{3.3}$$

$$\alpha < \beta \quad \texttt{means } \alpha \texttt{ is earlier than } \beta \texttt{ and } \beta \texttt{ is after } \alpha \tag{3.4}$$

$$\gamma \le \delta \quad \texttt{means } \delta \texttt{ is either } \gamma \texttt{ or later than } \gamma \tag{3.5}$$

Let lowercase letters $\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}$ denote declarations. $\mathtt{F(d)} = \beta$ means that the feature assignment of declaration `d` is β. We use $\perp$ to denote the *unknown* feature. $\mathtt{F(d)} = \perp$ means that the feature of `d` has not been assigned.

In the following, $\mathbb{D}$ denotes the domain of all program declarations, $\mathbb{DD}$ denotes a computable subset of $\mathbb{D}$, $\mathbb{F} = \{\alpha \ldots \omega\}$ is the domain of all features, and $\mathbb{P}$ is the domain of all assignment *possibilities*, *ie* $\mathbb{P} = \mathbb{F} \times \mathbb{D} \times \{\mathtt{fprivate}, \mathtt{fpublic}\}$. We explain possibilities later in this chapter. Finally, to permit declarations that are not assigned to features, we use the domain $\mathbb{FF} = \mathbb{F} \cup \{\perp\}$.

Table 3.1 contains the predicates of the rules and formulas that we will use; an explanation of each entry follows:

- `PD` is the set of declarations encountered in a program's CRG.
- `e` is a declaration encountered in a program's CRG.
- `parent(e)` is `e`'s container.
- `F(e)` is the feature assignment of declaration `e`. If unassigned, $\mathtt{F(e)} = \perp$.
- `eb(e)`, `lb(e)`, `FB(e)` define the feature bounds of `e`. `eb(e)` is the earliest feature to which `e` could be assigned, `lb(e)` is the latest feature, and the feature bounds of `e` is $\mathtt{FB(e)} = \{\mathtt{lb(e)} \ldots \mathtt{ub(e)}\}$.[7]
- `fprivate(e)` and `fpublic(e)` indicate the feature modifier that is assigned to `e`. Initially `e` is unassigned: `fprivate(e) = fpublic(e) = false`. A modifier assignment makes precisely one of `fprivate(e)` and `fpublic(e)` `true`.
- `RefFrom(e)` and `RefTo(e)` return the sets of declarations that `e` references and is referenced by. `AssignedRefFrom(e)` are declarations in `RefFrom(e)` that have been assigned to a feature. `AssignedRefTo(e)` are declarations in `RefTo(e)` that have been assigned to a feature.
- `AtLeastOneAssigned(`$\mathbb{DD}$`)` returns `true` if at least one declaration in $\mathbb{DD}$ is assigned to a feature. `NoneAssigned(`$\mathbb{DD}$`)` returns `true` if none of the declarations in $\mathbb{DD}$ are assigned to a feature.

- `method(e)`, `hook(e)`, `field(e)`, `container(e)`[5] indicate whether `e` is a method, hook, field, or container. Note that only one of `method(e)`, `field(e)`, `container(e)` is ever true, *eg*:[6]

$$\neg\,\texttt{method(e)} \;=\; \texttt{field(e)} \lor \texttt{container(e)} \tag{3.6}$$

Predicate	**Type**	**Meaning**
`PD`	$\mathbb{DD}$	Set of declarations in a program's CRG
`e`	$\mathbb{D}$	A program declaration
`parent(e)`	$\mathbb{D}$	A program declaration's container
`F(e)`	$\mathbb{FF}$	Assigned feature of `e`
`eb(e)`	$\mathbb{F}$	Earliest Bound of `e`
`lb(e)`	$\mathbb{F}$	Latest Bound of `e`
`FB(e)`	range($\mathbb{F}$)	Valid features in range for `e`, $\texttt{FB(e)} = \{\texttt{eb(e)} \ldots \texttt{lb(e)}\}$ [7]
`fprivate(e)`	Boolean	Modifier of `e` is `fprivate`
`fpublic(e)`	Boolean	Modifier of `e` is `fpublic`
`RefFrom(e)`	$\mathbb{DD}$	References made by `e` to other declarations
`AssignedRefFrom(e)`	$\mathbb{DD}$	Declarations referenced by `e` that have been assigned to a feature: $\texttt{AssignedRefFrom(e)} = \{\, \texttt{d} \mid \texttt{d} \in \texttt{RefFrom(e)} \land \big(\texttt{F(d)} \neq \perp\big) \}$
`RefTo(e)`	$\mathbb{DD}$	References from other declarations to `e`
`AssignedRefTo(e)`	$\mathbb{DD}$	Declarations that reference `e` that have been assigned to a feature: $\texttt{AssignedRefTo(e)} = \{\texttt{d} \mid \texttt{d} \in \texttt{RefTo(e)} \land \big(\texttt{F(d)} \neq \perp\big)\}$
`AtLeastOneAssigned`($\mathbb{DD}$)	Boolean	There is at least one declaration in $\mathbb{DD}$ that is assigned to a feature, $\exists \texttt{r}\big(\,\texttt{r} \in \mathbb{DD} \land \texttt{F(r)} \neq \perp\big)$
`NoneAssigned`($\mathbb{DD}$)	Boolean	None of the declarations in $\mathbb{DD}$ are assigned to a feature, $\forall \texttt{r}\big(\,\texttt{r} \in \mathbb{DD} \Rightarrow \texttt{F(r)} = \perp\big)$
`ByInference(e)`	Boolean	F(e) was assigned by inference
`ByInferenceDec(e)`	$\mathbb{D}$	Declaration that caused `e` to be assigned by inference. `e` references this declaration. $\texttt{ByInferenceDec(e)} \in \texttt{RefFrom(e)}$
`method(e)`	Boolean	`e` is a method
`hook(e)`	Boolean	`e` is a hook method
`field(e)`	Boolean	`e` is a field
`container(e)`	Boolean	`e` is a container
$\mathbb{P}$`(F,e,mod)`	Boolean	An assignment possibility composed of a feature, a declaration and a modifier
$\mathbb{P}$`(e)`	$\mathbb{P}$	All assignment possibilities for declaration `e`

Table 3.1: Basic Predicates

[5] Classes and packages are considered containers.
[6] $\texttt{exactlyOne}(\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n)$ is true if precisely one of the $\texttt{e}_i$ arguments is true.
[7] `eb(e)` and `lb(e)` will never be $\perp$. Their formulas defined in this chapter prevent this from happening.

The big picture for feature and modifier calculations is to determine the options that B^t_tF offers its users. When B^t_tF asks users to assign a feature and modifier to a declaration `e`, it displays via GUI buttons the set of all legally permissible (feature,modifier) pairs. Each pair is called a *possibility*. Figure 3.3 shows a snapshot of B^t_tF's GUI that lists 8 legal possibilities for some declaration `e` in a program that is to be partitioned into features `BASE`, `TABLE`, `CONSTRAINT` and `EVAL`. The possibilities in Figure 3.3 are $\mathbb{P}(\texttt{e})$:

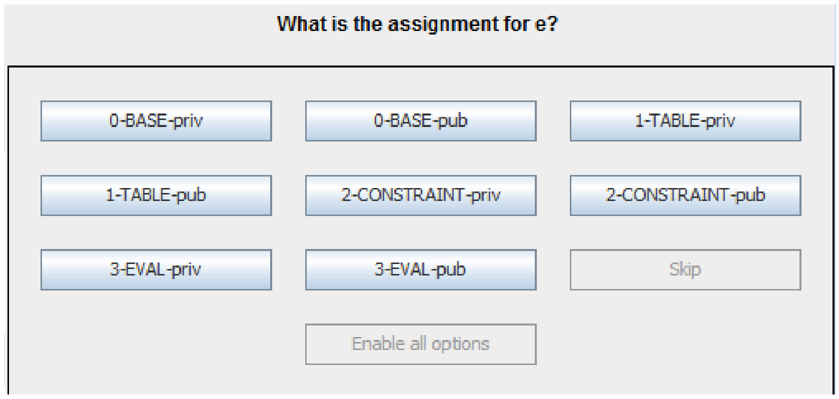


Figure 3.3: B^t_tF's Possibility Offerings for a Declaration `e`

$$\begin{aligned}\mathbb{P}(\texttt{e}) \;=\; &\big\{(\texttt{BASE},\texttt{e},\texttt{fprivate}),(\texttt{BASE},\texttt{e},\texttt{fpublic}),(\texttt{TABLE},\texttt{e},\texttt{fprivate}),(\texttt{TABLE},\texttt{e},\texttt{fpublic}),\\ &(\texttt{CONSTRAINT},\texttt{e},\texttt{fprivate}),(\texttt{CONSTRAINT},\texttt{e},\texttt{fpublic}),(\texttt{EVAL},\texttt{e},\texttt{fprivate}),\\ &(\texttt{EVAL},\texttt{e},\texttt{fpublic})\big\}.\end{aligned}$$

These are the legal fact assignments that B^t_tF allows users to make. The possibilities for one declaration `d` may be different than the possibilities for another declaration $\texttt{e} \neq \texttt{d}$.

At runtime, B^t_tF maintains the disjunction of invariants in Table 3.2. Invariant `(i)` is the initial value for `e`'s feature assignment, which is $\perp$. Invariant `(ii)` means when a user assigns `e` as `fprivate` of a feature, that feature exists and `e` cannot be `fpublic` too. Invariant `(iii)` means when a user assigns `e` as `fpublic` of a feature, that feature exists and `e` cannot be `fprivate` too. Invariant `(iv)` means B^t_tF inferred `e`'s feature assignment, then neither modifier is applied. Precisely one of `(i)` – `(iv)` always holds for every $\texttt{e} \in \texttt{PD}$.

With reference to `(iv)`, the primary goal of B^t_tF is to assign features to program declarations. If B^t_tF discovers that a declaration `d` can be assigned to only one feature ($\texttt{FB}(\texttt{d}) = [\eta...\eta]$), it assigns $\texttt{F}(\texttt{d}) = \eta$, and proceeds to the next declaration in the CRG. Declaration `d` does not have a feature modifier assigned to it because the modifier could be ultimately be either `fprivate` or `fpublic`.

Once a program is partitioned, only `(ii)`-`(iv)` apply, as no declarations satisfy `(i)`.

Finally, B^t_tF uses rules of the form:

$$\frac{\text{typed variable bindings and preconditions}}{}$$

#	Invariant	fprivate(e)	fpublic(e)
(i)	$\mathtt{F(e)} = \bot$	false	false
(ii)	$\mathtt{F(e)} \in \mathbb{F} \land \mathtt{fprivate(e)} \land \neg\mathtt{fpublic(e)}$	true	false
(iii)	$\mathtt{F(e)} \in \mathbb{F} \land \neg\mathtt{fprivate(e)} \land \mathtt{fpublic(e)}$	false	true
(iv)	$\mathtt{F(e)} \in \mathbb{F} \land \neg\mathtt{fprivate(e)} \land \neg\mathtt{fpublic(e)}$	false	false

Table 3.2: Invariant Feature and Modifier Combinations

$$\text{feature or modifier or possibility assignments using above variables} \tag{3.7}$$

That is, a set of typed variables are declared with constraints followed by feature or modifier assignments that reference these variables. Assignments may be conditional.

3.2.9 Hooks

A *hook* is a method that can reference declarations in future features. Only methods can be hooks; B^t_tF uses the following rule to recognize hooks:

$$\frac{\mathtt{method(m)} \land \left(\exists \mathtt{e} \left(\mathtt{e} \in \mathtt{RefFrom(m)} \land (\mathtt{F(m)} \neq \bot) \land (\mathtt{F(e)} \neq \bot) \land (\mathtt{F(m)} < \mathtt{F(e)})\right)\right)}{\mathtt{hook(m)}} \tag{3.8}$$

That is, if method `m` references `e`, and both `m` and `e` have been assigned to a feature, and `F(e)` is after than `F(m)` (*ie*, `e` is defined in a future layer than `m`), then `m` is a hook.

Let `h` be a hook. `h`, as the rest of declarations, can reference any declaration in the past that is `fpublic`, and also reference declarations in future features regardless of their modifier. In other words, `h` can reference a declaration `e` when the latter is:

1. `fpublic` regardless of its feature, or
2. `fprivate` and assigned to a feature at or after `F(h)`.

B^t_tF checks this invariant at runtime, which is expressed as following:

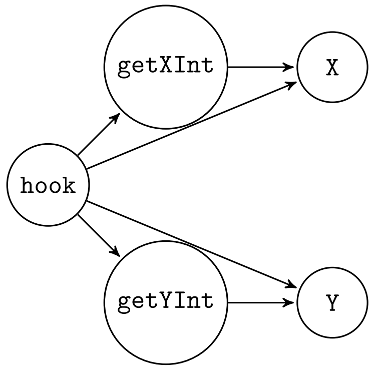
$$\frac{\mathtt{hook(h)}}{\forall \mathtt{e}\Big(\left(\mathtt{e} \in \mathtt{AssignedRefFrom(h)}\right) \Rightarrow \left(\mathtt{fpublic(e)} \lor \left(\mathtt{fprivate(e)} \Rightarrow \left(\mathtt{F(h)} \leq \mathtt{F(e)}\right)\right)\right)\Big)} \tag{3.9}$$

A class is a container. Members of a class require the class to exist before they are introduced. In a CRG, members reference their container. A classical method in a CRG references declarations in its body. A hook exhibits both classical method and container-like properties.

Figure 3.4a presents a hook method `hook`, which belongs to feature β. (In the following, remember $\alpha < \beta < \delta$). Declarations in line 4: `Y` and `getYInt` belong to a future feature δ. Declarations `Y` and `getYInt` use `hook` as a container. `hook` also exhibits classical method behavior as it references declarations `getXInt` and `X` which belong to past feature α. This means they were defined earlier than the `hook` method.

```
int hook(X op) {
  //block added by another feature
  if(op.getXInt() == 1){
    return new Y.getYInt(op);
  }
  //original code
  return op.getXInt();
}
```

(a)



(b)

F(getXInt) = α
F(X) = α
F(hook) = β
F(getYInt) = δ
F(Y) = δ

(c)

Figure 3.4: Hook Example

Note: We decided that field IEs cannot be hooks, even if its IE has inlined the contents of a hook method. Consider Figure 3.5a that presents field `var` calling hook `init`. Figure 3.5b presents code with the same functionality as Figure 3.5a, in this case it is an IE with the hook method inlined. So the question is: besides methods, could field IEs be hooks? We note that in OO languages, only classes can be extended and only methods can be overridden, not field IEs. With this observation, and to avoid additional (and perhaps unnecessary complexity), we discarded the possibility of having hook fields (or rather field IEs).

```
X var = init();
X init() { //init is a hook
  return new X().increment();
}
```

(a)

```

X var = new X().increment();


```

(b)

Figure 3.5: A Field Initialization Expression

3.3 Feature Bounds and Possibilities Calculation

Layers are fundamental to our feature partitioning approach. As features are expected to behave as layers, it is necessary to establish a feature bounds for each declaration. A *feature bounds* is a range of features that are valid assignments for a declaration. Feature bounds are derived by equating features with layers. Doing so guarantees the type correctness of each layer that are built in a step-wise (incremental) manner.

B$^{t}_{t}$F is a Truth Maintenance System [38]. It collects and infers facts, guaranteeing they are consistent. Assigning declarations to features imposes constraints on other declarations to avoid invalid partitioning (*ie* inconsistencies). Feature bounds are important to prevent erroneous assignments. Also, since feature bounds may reduce the available feature options, they may simplify a user's work to feature-partition a legacy application.

3.3.1 Importance and Definition of Feature Bounds

Figure 3.6 is useful to explain the importance of feature bounds to prevent wrong assignments of declarations to features. This CRG contains three declarations `a`, `b` and `c`; `a` has been assigned to feature β, and `b` has been assigned to δ. The edges are $\{\texttt{a},\texttt{b}\} \rightarrow \texttt{c}$. For `c` to be visible to both `a` and `b`, it has to be assigned to a feature introduced at or before β.[8,10] Given this, the only valid feature options for `c` are α and β.

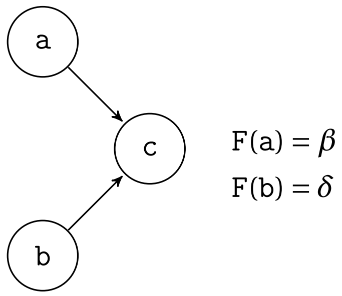


Figure 3.6: Risk of wrong assignment

Feature bounds `FB(e)` = {`eb(e)` ... `lb(e)`} of a declaration `e` determines the range of features to which `e` can be assigned. `eb(e)` is the earliest valid feature; `lb(e)` is the latest valid feature. `eb(e)` establishes how far in the past `e` could be assigned, and `lb(e)` sets how far in the future it could be assigned without violating existing constraints.

`eb(e)` is always at or after the `base` feature α, `lb(e)` is always at or before the latest feature ω. Thus:

$$\alpha \leq \texttt{eb(e)} \leq \texttt{F(e)} \leq \texttt{lb(e)} \leq \omega$$

Bounds calculations vary depending on existing assignments and declaration type. At the beginning of the `BttF` partitioning process, no feature assignment has been made. As assignments are made, there is more information to restrict bounds.

Importance of `RefTo(e)` and `RefFrom(e)`

Hooks complicate many things, but not bounds calculation. The reason is that bounds are computed *before* a method is determined to be a hook. So keep this in mind as we proceed.

To calculate `eb(e)` we use the references made by `e`, `RefFrom(e)`. For `e` to reference declarations in `RefFrom(e)`, `e` needs to be introduced at or after the latest of the features of the declarations in `RefFrom(e)`.[9]

To calculate `lb(e)` we use the references made to `e`, `RefTo(e)`. For declarations in `RefTo(e)` to reference `e`, `e` needs to be introduced at or before the earliest of the features of the dec-

[8] `earliest`$(\beta,\delta) = \beta$ where $\beta \leq \delta$.

[9] A declaration (other than a hook) can only reference declarations at or before it was defined.

larations in `RefTo(e)`.[10] When `e` is a method, we also consider the `fprivate` declarations it references for calculating `lb`.

In the possibility calculation process, bounds are obtained first, then modifiers. The following sections explain the rules for calculating feature bounds and modifiers, using assignment information of `e`'s surrounding CRG declarations.

3.3.2 Possibility Calculations Process

Let `e` be a declaration. We first describe how to assign bounds to `e`, then how to determine the possibilities for `e`.

Step 1. Obtain Earliest Bound `eb(e)`

`eb(e)` is calculated depending on `e`'s declaration type. There are two cases: when `e` is not a method and when `e` is.

Case 1. `e` is not a method. The IE of a field can reference classes, methods and other fields.[11] Containers can only reference other containers. Both fields and containers always follow a layered design, they can only reference declarations that have been already introduced.

Let `AssignedRefFrom(e)` be the set of declarations in `RefFrom(e)` that are assigned to a feature:

$$\mathtt{AssignedRefFrom(e)} = \{\, \mathtt{r} \mid \mathtt{r} \in \mathtt{RefFrom(e)} \wedge \mathtt{F(r)} \neq \perp \,\} \tag{3.10}$$

For `e` to reference declarations in `AssignedRefFrom(e)`, `e` must be introduced at or after the latest of the features of declarations in `AssignedRefFrom(e)`. If `AssignedRefFrom(e)` is empty, then `eb(e)` is α, the base feature.[12]

$$\frac{\neg\mathtt{method(e)}}{\mathtt{eb(e)} = \begin{cases} \mathtt{latest}\{\, \mathtt{F(r)} : \mathtt{r} \in \ \mathtt{AssignedRefFrom(e)} \,\}, & \text{if } \mathtt{AssignedRefFrom(e)} \neq \emptyset \\ \alpha, & \text{otherwise} \end{cases}} \tag{3.11}$$

Case 2. `e` is a method. Since `e` might be a hook, it could be assigned to a feature that is earlier than the features of declarations in `RefFrom(e)`, but *not* earlier than its own container. In other words, `e` can be assigned as early as the feature of its own container. If its

[10] A declaration can be referenced at or after it has been defined, except if it is being referenced by a hook.

[11] Of course, a special case is that there is no IE, so a field declaration only references its declared type.

[12] α is the earliest possible feature, if none of the assigned declarations references `e`, then the earliest bound of `e` is α.

container's feature is unknown, then `eb(e)` is the earliest feature α:

$$\frac{\texttt{method(e)}}{\texttt{eb(e)} = \begin{cases} \texttt{F(parent(e))}, & \text{if } \texttt{parent(e)} \neq \perp \wedge \texttt{F(parent(e))} \neq \perp \\ \alpha, & \text{otherwise} \end{cases}} \tag{3.12}$$

Step 2. Obtain Latest Bound `lb(e)`

`lb(e)` is calculated by a single formula. However, the value for a term used in the formula depends on `e`'s type. This term is `L(e)`, it represents the set of declarations that follow a layered design and impose a constraint on `e`'s feature assignment. *Key point: because methods may be hooks, they do not follow a 'classical' layered design, and thus do not influence* `lb(e)` *calculations.* The earliest element in `L(e)` determines `lb(e)`. The following paragraphs explain how `L(e)` is computed.

Case 1. `e` is a container. `L(e)` is the set of declarations that reference container `e` or are contained in `e`. The only methods being considered here are those in `e`.

$$\texttt{L(e)} = \{\texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \big(\neg\,\texttt{method(r)} \vee \big(\texttt{method(r)} \wedge (\texttt{parent(r)} = \texttt{e})\big)\big)\} \tag{3.13}$$

Case 2. `e` is a field. `L(e)` is the set of fields[13] that reference `e`. Methods that reference `e` do not influence `lb(e)` as they could be hooks.

$$\texttt{L(e)} = \{\texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \neg\,\texttt{method(r)}\} \tag{3.14}$$

Case 3. `e` is a method. `e` can be referenced by methods and by fields, the latter always follow a layered design and require `e` to be introduced at or before the earliest of them. In addition, since `e` might be a hook, it could reference `fprivate` declarations in the present or future, but not it the past. In other words, `e` must be introduced at or before the earliest of the `fprivate` declarations that it references.

`L(e)` is the set of fields that reference `e` union the set of `fprivate` declarations that `e` references.

$$\texttt{L(e)} = \{\texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \neg\,\texttt{method(r)}\} \cup \{\texttt{q} \mid \texttt{q} \in \texttt{RefFrom(e)} \wedge \texttt{fprivate(q)}\} \tag{3.15}$$

Formula. Declarations in `L(e)` require `e` to be introduced at or earlier than them. Thus `lb(e)` is determined by the earliest assigned declaration in `L(e)`. For this and following sections let `AssignedL(e)` be the set of declarations in `L(e)` that have been already assigned to a feature.

$$\texttt{AssignedL(e)} = \{\texttt{r} \mid \texttt{r} \in \texttt{L(e)} \wedge \texttt{F(r)} \neq \perp\} \tag{3.16}$$

[13] Field IE's.

e has to be assigned to a feature at or before the earliest of the features of AssignedL(e). If there are no declarations in AssignedL(e), then the lb(e) is ω, the latest feature:

$$\frac{\texttt{TRUE}}{\texttt{lb(e)} = \begin{cases} \texttt{earliest}\{\,\texttt{F(r)} : \texttt{r} \in \texttt{AssignedL(e)}\,\}, & \text{if } \texttt{AssignedL(e)} \neq \emptyset \\ \omega, & \text{otherwise} \end{cases}} \tag{3.17}$$

Step 3. Obtain Possibilities

Once the feature bounds for e have been obtained, we can calculate the valid modifiers for e to obtain the set of possibilities to present to a user. There are only two cases:

Case A. `fpublic` possibilities. Determining `fpublic` possibilities is easy: e can be `fpublic` on any of the features in bounds for e.

$$\frac{\texttt{TRUE}}{\forall \texttt{F}\ \big(\texttt{F} \in \texttt{FB(e)} \Rightarrow \mathbb{P}(\texttt{F,e,fpublic})\big)} \tag{3.18}$$

Case B. `fprivate` possibilities. There are four independent cases for calculating `fprivate` possibilities. The `fprivate` modifier affects a declaration's *visibility*. If e is `fprivate` to feature F, only declarations in feature F can reference e.[14] For this reason, to determine e's `fprivate` possibilities we consider the available information about declarations in RefTo(e). From RefTo(e) we obtain two subsets D(e) and M(e). This means:

$$\texttt{RefTo(e)} = \texttt{D(e)} \cup \texttt{M(e)} \tag{3.19}$$

D(e) contains fields and containers that reference e and also all declarations contained in e.[15] M(e) represents the set of methods that reference e and are not contained in e.[16] Declarations in M(e) can reference e regardless of its modifier.

$$\texttt{D(e)} = \{\,\texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \big(\neg\,\texttt{method(r)} \vee \big(\texttt{method(r)} \wedge (\texttt{parent(r)} = \texttt{e})\big)\big)\,\} \tag{3.20}$$

$$\texttt{M(e)} = \{\,\texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \texttt{method(r)} \wedge (\texttt{parent(r)} \neq \texttt{e})\,\} \tag{3.21}$$

Let AssignedD(e) and AssignedM(e) be the subset of declarations in D(e) and M(e), respectively, that have been already assigned to a feature.

$$\texttt{AssignedD(e)} = \{\,\texttt{r} \mid \texttt{r} \in \texttt{D(e)} \wedge \texttt{F(r)} \neq \bot\,\} \tag{3.22}$$

$$\texttt{AssignedM(e)} = \{\,\texttt{r} \mid \texttt{r} \in \texttt{M(e)} \wedge \texttt{F(r)} \neq \bot\,\} \tag{3.23}$$

Table 3.3 contains eight disjoint scenarios that can be encountered in D(e) and M(e) and the case that is used to calculate `fprivate` possibilities. The union of the possibilities of these disjoint scenarios equals all `fprivate` possibilities.[17]

[14] `fprivate` does not affect the references a declaration can make.

[15] Assuming e is a container.

[16] Assuming e is a container.

[17] D can be *empty*, *none assigned*, or *at least one assigned*. So too can M. This yields the 9 scenarios listed in

#	Scenario for `fprivate` calculation	Case
No declarations in either `D(e)` ***or*** `M(e)`		
(i)	$\big(\mathtt{D(e)} = \emptyset\big) \wedge \big(\mathtt{M(e)} = \emptyset\big)$	1
Only declarations in `D(e)` ***- none are assigned***		
(ii)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} = \emptyset\big) \wedge \mathtt{NoneAssigned(D(e))}$	1
Only declarations in `D(e)` ***– at least one assigned***		
(iii)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} = \emptyset\big) \wedge \mathtt{AtLeastOneAssigned(D(e))}$	2
Only declarations in `M(e)` ***– none are assigned***		
(iv)	$\big(\mathtt{D(e)} = \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{NoneAssigned(M(e))}$	1
Only declarations in `M(e)` ***– at least one assigned***		
(v)	$\big(\mathtt{D(e)} = \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{AtLeastOneAssigned(M(e))}$	3
Declarations in `D(e)` ***and*** `M(e)` ***– none are assigned***		
(vi)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{NoneAssigned(D(e))} \wedge \mathtt{NoneAssigned(M(e))}$	1
Declarations in `D(e)` ***and*** `M(e)` ***– at least one of the first assigned, none of the second assigned***		
(vii)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{AtLeastOneAssigned(D(e))} \wedge \mathtt{NoneAssigned(M(e))}$	2
Declarations in `D(e)` ***and*** `M(e)` ***– none of the first assigned, at least one of the second assigned***		
(viii)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{NoneAssigned(D(e))} \wedge \mathtt{AtLeastOneAssigned(M(e))}$	3
Declarations in `D(e)` ***and*** `M(e)` ***– at least one of each assigned***		
(ix)	$\big(\mathtt{D(e)} \neq \emptyset\big) \wedge \big(\mathtt{M(e)} \neq \emptyset\big) \wedge \mathtt{AtLeastOneAssigned(D(e))} \wedge \mathtt{AtLeastOneAssigned(M(e))}$	4

Table 3.3: Scenarios to Consider for `fprivate` possibilities

Case 1 (Scenarios (i),(ii),(iv),(vi)). If none of the declarations in `D(e)` and `M(e)` have been assigned, `e` can be `fprivate` of any feature in bounds.

$$\frac{\forall \mathtt{r}\ \big(\ \mathtt{r} \in \big(\ \mathtt{D(e)}\ \cup\ \mathtt{M(e)}\big)\ \Rightarrow\ \mathtt{F(r)}\ = \perp\ \big)}{\forall \mathtt{F}\ \big(\mathtt{F} \in \mathtt{FB(e)} \Rightarrow \mathbb{P}(\mathtt{F},\mathtt{e},\mathtt{fprivate})\ \big)} \tag{3.24}$$

Note: The CRG in Figure 3.7 illustrates Case 1. We have declarations `a,b,c,d` that reference `e`. None of `a,b,c,d` have been assigned to a feature. Then, `e` can be `fprivate` of any feature in `FB(e)`.[18]

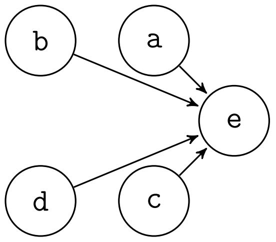
Case 2 (Scenario (iii)). If only declarations in `D(e)` have been assigned[19], and they all are assigned to the same feature `F`, and `F` is a feature in bounds for `e`, then `e` can be `fprivate` of `F`.

$$\frac{\begin{gathered}\exists \mathtt{F}\Big(\ \mathtt{F} \in \mathtt{FB(e)}\ \wedge\ \forall \mathtt{r}\big(\ \mathtt{r}\ \in\ \mathtt{AssignedD(e)}\ \Rightarrow \mathtt{F(r)} = \mathtt{F}\big)\Big)\\ \wedge\ \big(\ \mathtt{AssignedM(e)} = \emptyset\ \big)\end{gathered}}{\mathbb{P}(\mathtt{F},\mathtt{e},\mathtt{fprivate})} \tag{3.25}$$

Table 3.3.

[18] Of course, if a user selects `e` to be `fprivate`, doing so imposes many constraints on `a,b,c,d`. But this is OK, as it is a *possibility*.

[19] There are no methods that reference `e`, or none of them have been assigned.

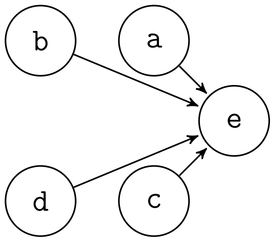


$$\begin{aligned}\texttt{method(a)} \wedge \texttt{F(a)} &= \bot\\ \texttt{method(b)} \wedge \texttt{F(b)} &= \bot\\ \texttt{field(c)} \wedge \texttt{F(c)} &= \bot\\ \texttt{container(d)} \wedge \texttt{F(d)} &= \bot\end{aligned}$$

Figure 3.7: `fprivate` possibilities. Case 1.

Note: The CRG in Figure 3.8 illustrates Case 2. We have declarations a, b, c, d that reference e. c is a field, and d is a container, and both have been assigned to feature δ. Declaration a is a method contained in e and has been assigned to feature δ, and b is a method that references e and has not been assigned to a feature. In this example, assuming $\delta \in \texttt{FB(e)}$, a possibility for e is:

$$\mathbb{P}(\delta, \texttt{e}, \texttt{fprivate})$$

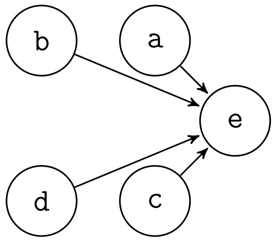


$$\begin{aligned}\texttt{method(a)} \wedge \texttt{parent(a)} = \texttt{e} \wedge \texttt{F(a)} &= \delta\\ \texttt{method(b)} \wedge \texttt{parent(b)} \neq \texttt{e} \wedge \texttt{F(b)} &= \bot\\ \texttt{field(c)} \wedge \texttt{F(c)} &= \delta\\ \texttt{container(d)} \wedge \texttt{F(d)} &= \delta\end{aligned}$$

Figure 3.8: `fprivate` possibilities. Case 2.

Case 3 (Scenarios (v),(viii)). If there are no declarations in D(e) or none of them have been assigned, and at least one method in M(e) has been assigned, and the latest of their features is in bounds for e, then e can be `fprivate` of the range of features in bounds that are larger or equal than it. This does not affect declarations in M(e) because all of them have the ability to reference `fprivate` declarations in the future.

$$\frac{\begin{array}{c}\big(\texttt{AssignedD(e)} = \emptyset\big)\\ \wedge \big(\texttt{AssignedM(e)} \neq \emptyset\big) \wedge \big(\texttt{latest}\big\{\texttt{F(r)} : \texttt{r} \in \texttt{AssignedM(e)}\big\} \in \texttt{FB(e)}\big)\end{array}}{\forall \texttt{F}\ \big(\texttt{F} \in \texttt{FB(e)} \vee \texttt{F} \geq (\texttt{latest}\big\{\texttt{F(r)} : \texttt{r} \in \texttt{AssignedM(e)}\big\}) \Rightarrow \mathbb{P}(\texttt{F}, \texttt{e}, \texttt{fprivate})\big)} \quad (3.26)$$



$$\begin{aligned} \texttt{method(a)} \wedge \texttt{parent(a)} \neq \texttt{e} \wedge \texttt{F(a)} = \beta \\ \texttt{method(b)} \wedge \texttt{parent(b)} \neq \texttt{e} \wedge \texttt{F(b)} = \gamma \\ \texttt{field(c)} \wedge \texttt{F(c)} = \perp \\ \texttt{container(d)} \wedge \texttt{F(d)} = \perp \end{aligned}$$

Figure 3.9: `fprivate` possibilities. Case 3.

Note: The CRG in Figure 3.9 illustrates Case 3. We have declarations a, b, c, d that reference e. Declarations a and b are methods that reference e and are not contained in it. a has been assigned to β, and b has been assigned to γ. c is a field, and d is a container, and neither have been assigned to a feature. In this example, assuming $\gamma \in \texttt{FB(e)}$, a possibility for e is:[20]

$$\mathbb{P}(\gamma, \texttt{e}, \texttt{fprivate})$$

Case 4 (Scenario (ix)). There are assigned declarations in both D(e) and M(e). AssignedD(e) contains assigned declarations in D(e). AssignedM(e) contains assigned declarations in M(e). If all declarations in AssignedD(e) have the same feature F and that feature is at or after the latest feature of declarations[21] in AssignedM(e), then e can be `fprivate` of F.

$$\frac{\begin{gathered}\big(\texttt{AssignedD(e)} \neq \emptyset\big) \wedge \big(\texttt{AssignedM(e)} \neq \emptyset\big) \\ \wedge\, \exists \texttt{F}\Big(\texttt{F} \in \texttt{FB(e)} \wedge \forall \texttt{r}\big(\texttt{r} \in \texttt{AssignedD(e)} \Rightarrow \texttt{F(r)} = \texttt{F}\big) \\ \wedge \big(\texttt{latest}\{\texttt{F(r)} : \texttt{r} \in \texttt{AssignedM(e)}\} \leq \texttt{F}\big)\Big)\end{gathered}}{\mathbb{P}(\texttt{F}, \texttt{e}, \texttt{fprivate})} \tag{3.27}$$

Note: The CRG in Figure 3.10 illustrates Case 4. We have declarations a, b, c, d that reference e. Declarations a and b are methods that reference e and are not contained in e. a has been assigned to β, and b has been assigned to γ. c is a field, and d is a container, and both have been assigned to feature δ. In this example, assuming $\delta \in$ FB(e), possibility for e is:

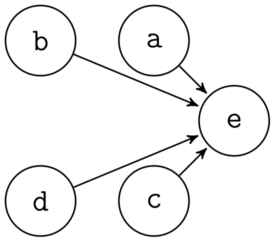
$$\mathbb{P}(\delta, \texttt{e}, \texttt{fprivate})^{22}$$

The following subsection explain how a possibility is validated, using two attributes: Time Consistency and Visibility.

[20] $\texttt{latest}(\beta,\gamma) = \gamma$.

[21] AssignedM(e) contains methods. These can reference any declaration in the future regardless of the declaration's modifier.

[22] $(\texttt{latest}(\beta,\gamma) = \gamma) \wedge (\gamma < \delta)$.



$$\texttt{method(a)} \wedge \texttt{parent(a)} \neq \texttt{e} \wedge \texttt{F(a)} = \beta$$
$$\texttt{method(b)} \wedge \texttt{parent(b)} \neq \texttt{e} \wedge \texttt{F(b)} = \gamma$$
$$\texttt{field(c)} \wedge \texttt{F(c)} = \delta$$
$$\texttt{container(d)} \wedge \texttt{F(d)} = \delta$$

Figure 3.10: `fprivate` possibilities. Case 4.

3.3.3 Possibility Validation

A possibility is valid if all affected edges in a CRG maintain these two properties: Time Consistency and Visibility. Let `src(ed)` be the source declaration of edge `ed`, and `dst(ed)` be destination declaration of `ed`. Edge `ed` is `src(ed)` → `dst(ed)`.

Time Consistency. An edge `ed` is *time consistent* if `dst(ed)` is assigned at or before the same time than `src(ed)`. Except if `src(ed)` is a method,[23] then this time order requirement is not necessary, as methods can become hooks.

$$\texttt{timeConsistency(ed)} = \big(\texttt{F(src(ed))} \geq \texttt{F(dst(ed))}\big) \vee \texttt{method(src(ed))} \tag{3.28}$$

Visibility. If `dst(ed)` is `fprivate`, then an edge `ed` is *visibility consistent only if both* `dst(ed)` *and* `src(ed)` *are in the same feature*. Again, an exception occurs if `src(ed)` is a method, as it can reference any `fprivate` declaration in the future.

$$\begin{aligned}\texttt{visibility(ed)} = &\Big(\texttt{fprivate(dst(ed))} \wedge \Big(\big(\texttt{F(src(ed))} = \texttt{F(dst(ed))}\big) \vee \texttt{method(src(ed))}\Big)\Big)\\ &\vee \neg\texttt{fprivate(dst(ed))}\end{aligned} \tag{3.29}$$

We do not have a mathematical proof for previously stated properties for possibilities calculation. Thus these invariants are maintained by B$^{\mathtt{t}}_{\mathtt{t}}$F to double-check there are no contradictions on feature assignments.

The following section provide examples that show how possibility calculation and validation rules are applied.

3.3.4 Example: Assigning a method

We are to assign `method_m` to a feature. The CRG in Figure 3.11 contains the surrounding declarations of `method_m`.

Reference edges are:

[23] Suppose `src(ed)` is a declaration to assign and it is a method. Because of this scenario, in the formula we say method and not hook. At this point its assignment has not yet been made, thus it is not possible to determine if it is a hook.

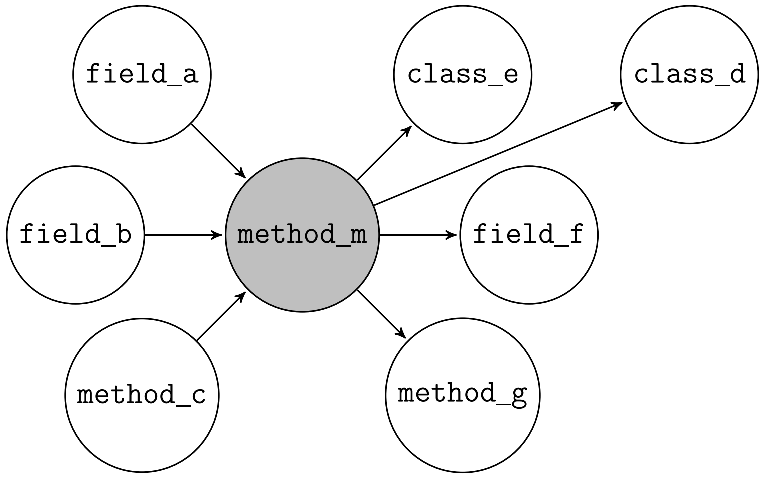


Figure 3.11: Example: Assigning a method.

- {`field_a`,`field_b`,`method_c`} → `method_m`
- `method_m` → {`class_e`,`field_f`,`method_g`}.

Containment edge is:

- `method_m` → `class_d`. This means that `parent(method_m) = class_d`.

The type and feature-modifier assignments for each declaration is listed below:

`field(field_a)`	$\wedge$	`F(field_a)` $=\epsilon$	$\wedge$	`fprivate(field_a)`
`field(field_b)`	$\wedge$	`F(field_b)` $=\perp$	$\wedge$	`true`
`method(method_c)`	$\wedge$	`F(method_c)` $=\delta$	$\wedge$	`fpublic(method_c)`
`method(method_m)`	$\wedge$	`F(method_m)` $=\perp$	$\wedge$	`true`
`container(class_d)`	$\wedge$	`F(class_d)` $=\beta$	$\wedge$	`fpublic(class_d)`
`container(class_e)`	$\wedge$	`F(class_e)` $=\alpha$	$\wedge$	`fpublic(class_e)`
`field(field_f)`	$\wedge$	`F(field_f)` $=\epsilon$	$\wedge$	`fprivate(field_f)`
`method(method_g)`	$\wedge$	`F(method_g)` $=\beta$	$\wedge$	`fpublic(method_g)`

Step 1. Obtain `eb(method_m)`

The earliest feature bound of a method is determined by its container. `method_m`'s container is `class_d`, the latter has been assigned to feature β, then `eb(method_m)` is β.

Step 2. Obtain `lb(method_m)`

To obtain `lb(method_m)` we need to obtain the declarations that belong to `L(method_m)`. These are the set of fields that reference `method_m` union with the set of `fprivate` declarations that `e` references. `AssignedL(method_m)` contains those declarations in `L(method_m)` that have been assigned to a feature.

$$\texttt{AssignedL(method\_m)} = \{\texttt{field\_a}, \texttt{field\_f}\}$$

`lb(method_m)` is determined by the earliest feature of declarations in `AssignedL(method_m)`, which is ϵ, therefore, `lb(method_m)` is ϵ. This gives us the following features in bounds for `method_m`: $\beta < \gamma < \delta < \epsilon$.

$$\texttt{FB(method\_m)} = [\,\beta \ldots \epsilon\,]$$

Step 3.1. Calculate `fpublic` **Possibilities**

`method_m` can be `fpublic` of any feature in bounds:

$$\mathbb{P}(\beta, \texttt{method\_m}, \texttt{fpublic}) \tag{3.30}$$
$$\mathbb{P}(\gamma, \texttt{method\_m}, \texttt{fpublic}) \tag{3.31}$$
$$\mathbb{P}(\delta, \texttt{method\_m}, \texttt{fpublic}) \tag{3.32}$$
$$\mathbb{P}(\epsilon, \texttt{method\_m}, \texttt{fpublic}) \tag{3.33}$$

Step 3.2. Calculate `fprivate` **Possibilities**

The declarations in `AssignedD(method_m)` and `AssignedM(method_m)` are listed next.

$$\texttt{AssignedD(method\_m)} = \{\texttt{field\_a}\}$$
$$\texttt{AssignedM(method\_m)} = \{\texttt{method\_c}\}$$

Now, we calculate the possibilities for `method_m`. The union of all possibilities' rules outcomes, gives the complete set of possibilities to present to a user.

There are four cases to determine all `fprivate` possibilities.

Case 1. Since we have declarations in both `AssignedD(method_m)` and `AssignedM(method_m)`, this case does not apply.

Case 2. The same explanation for Case 1 applies here. This case does not apply.

Case 3. The same explanation for Case 1 applies here. This case does not apply.

Case 4. We have declarations in both `AssignedD(method_m)` and `AssignedM(method_m)`. The field in `AssignedD(method_m)` has feature ϵ. The method in `AssignedM(method_m)`: `method_c`, is assigned to feature δ. Since $\delta < \epsilon$, then `method_c` can be `fprivate` of ϵ:

$$\mathbb{P}(\epsilon,\texttt{method\_m},\texttt{fprivate}) \tag{3.34}$$

Possibilities Validation

Tables 3.4 to 3.8 show the validation of the obtained five possibilities. Each table lists the edges among assigned declarations,[24] the feature dependency (FD) of the edge, and validates both Time Consistency and Visibility properties of the edge. As can be observed, all the edges for all possibilities are valid. If a user selects possibility (3.30), (3.31) or (3.32), B^t_tF will identify `method_m` as a hook.

Edge e	FD(e)	Time Consistency	Visibility
field_a → method_m	$\epsilon \to \beta$	TRUE, $\epsilon \geq \beta$	TRUE, ¬fprivate(method_m)
method_c → method_m	$\delta \to \beta$	TRUE, $\delta \geq \beta$	TRUE, ¬fprivate(method_m)
method_m → class_d	$\beta \to \beta$	TRUE, $\beta \geq \beta$	TRUE, ¬fprivate(class_d)
method_m → class_e	$\beta \to \alpha$	TRUE, $\beta \geq \alpha$	TRUE, ¬fprivate(class_e)
method_m → field_f	$\beta \to \epsilon$	TRUE, method(method_m)	TRUE, method(method_m) ∨ fprivate(field_f)
method_m → method_g	$\beta \to \beta$	TRUE, $\beta \geq \beta$	TRUE, ¬fprivate(method_g)

Table 3.4: Example 1. Validation of (3.30).

Edge e	FD(e)	Time Consistency	Visibility
field_a → method_m	$\epsilon \to \gamma$	TRUE, $\epsilon \geq \gamma$	TRUE, ¬fprivate(method_m)
method_c → method_m	$\delta \to \gamma$	TRUE, $\delta \geq \gamma$	TRUE, ¬fprivate(method_m)
method_m → class_d	$\gamma \to \beta$	TRUE, $\gamma \geq \beta$	TRUE, ¬fprivate(class_d)
method_m → class_e	$\gamma \to \alpha$	TRUE, $\gamma \geq \alpha$	TRUE, ¬fprivate(class_e)
method_m → field_f	$\gamma \to \epsilon$	TRUE, method(method_m)	TRUE, method(method_m) ∨ fprivate(field_f)
method_m → method_g	$\gamma \to \beta$	TRUE, $\gamma \geq \beta$	TRUE, ¬fprivate(method_g)

Table 3.5: Example 1. Validation of (3.31).

[24] Surrounding declarations of `method_m` that have not been assigned are not considered because they did not impose any constraint.

Edge e	FD(e)	Time Consistency	Visibility
field_a → method_m	$\epsilon \rightarrow \delta$	TRUE, $\epsilon \geq \delta$	TRUE, ¬fprivate(method_m)
method_c → method_m	$\delta \rightarrow \delta$	TRUE, $\delta \geq \delta$	TRUE, ¬fprivate(method_m)
method_m → class_d	$\delta \rightarrow \beta$	TRUE, $\delta \geq \beta$	TRUE, ¬fprivate(class_d)
method_m → class_e	$\delta \rightarrow \alpha$	TRUE, $\delta \geq \alpha$	TRUE, ¬fprivate(class_e)
method_m → field_f	$\delta \rightarrow \epsilon$	TRUE, method(method_m)	TRUE, method(method_m) ∨ fprivate(field_f)
method_m → method_g	$\delta \rightarrow \beta$	TRUE, $\delta \geq \beta$	TRUE, ¬fprivate(method_g)

Table 3.6: Example 1. Validation of (3.32).

Edge e	FD(e)	Time Consistency	Visibility
field_a → method_m	$\epsilon \rightarrow \epsilon$	TRUE, $\epsilon \geq \epsilon$	TRUE, ¬fprivate(method_m)
method_c → method_m	$\delta \rightarrow \epsilon$	TRUE, method(method_c)	TRUE, ¬fprivate(method_m)
method_m → class_d	$\epsilon \rightarrow \beta$	TRUE, $\epsilon \geq \beta$	TRUE, ¬fprivate(class_d)
method_m → class_e	$\epsilon \rightarrow \alpha$	TRUE, $\epsilon \geq \alpha$	TRUE, ¬fprivate(class_e)
method_m → field_f	$\epsilon \rightarrow \epsilon$	TRUE, $\epsilon \geq \epsilon$	TRUE, fprivate(field_f)
method_m → method_g	$\epsilon \rightarrow \beta$	TRUE, $\epsilon \geq \beta$	TRUE, ¬fprivate(method_g)

Table 3.7: Example 1. Validation of (3.33).

3.4 B$^{t}_{t}$F Inferences and Examples

B$^{t}_{t}$F performs inferences, described below, in five scenarios. The first and second scenarios occur when a user provides a fact that indicates a declaration is fprivate. The third applies to declarations assigned to the latest feature of a feature model. The fourth occurs when only one feature is in bounds for a declaration. The fifth takes care of declarations that only reference their containers and are not referenced by other declarations.

3.4.1 Inference 1: fprivate Fields and Methods

If e is a field or a method, and it is assigned fprivate of a feature F, declarations in RefTo(e) that have not been assigned yet, are also automatically assigned to F. Declarations assigned by B$^{t}_{t}$F do not receive a modifier, because there is no way to infer it and because B$^{t}_{t}$F's main goal is to obtain feature assignments, not modifiers.

For this and the following sections, let NoAssignedRefTo(e) be the set of declarations in RefTo(e) that have not been assigned to a feature.

$$\texttt{NoAssignedRefTo(e)} = \{\, \texttt{r} \mid \texttt{r} \in \texttt{RefTo(e)} \wedge \texttt{F(r)} = \bot \,\} \tag{3.35}$$

If e is assigned fprivate of a feature F, then declarations in NoAssignedRefTo(e) will also be assigned to F.

Edge e	**FD(e)**	**Time Consistency**	**Visibility**
field_a → method_m	$\epsilon \rightarrow \epsilon$	TRUE, $\epsilon \geq \epsilon$	TRUE, fprivate(method_m) $\wedge\ \epsilon = \epsilon$
method_c → method_m	$\delta \rightarrow \epsilon$	TRUE, method(method_c)	TRUE, fprivate(method_m) $\wedge$ method(method_c)
method_m → class_d	$\epsilon \rightarrow \beta$	TRUE, $\epsilon \geq \beta$	TRUE, ¬fprivate(class_d)
method_m → class_e	$\epsilon \rightarrow \alpha$	TRUE, $\epsilon \geq \alpha$	TRUE, ¬fprivate(class_e)
method_m → field_f	$\epsilon \rightarrow \epsilon$	TRUE, $\epsilon \geq \epsilon$	TRUE, fprivate(field_f)
method_m → method_g	$\epsilon \rightarrow \beta$	TRUE, $\epsilon \geq \beta$	TRUE, ¬fprivate(method_g)

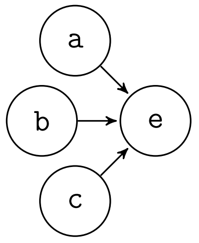
Table 3.8: Example 1. Validation of (3.34).

$$\frac{\big(\texttt{method(e)} \vee \texttt{field(e)}\big) \wedge \texttt{fprivate(e)} \wedge \texttt{F(e)} = \texttt{F}}{\forall \texttt{r}\ \big(\texttt{r} \in \texttt{NoAssignedRefTo(e)} \Rightarrow\ (\texttt{F(r)} = \texttt{F})\big)} \tag{3.36}$$

Consider CRG in Figure 3.12. Let's say that e is a field. Declarations a, b and c reference e. a and b have not been assigned to a feature, therefore they belong to NoAssignedRefTo(e).

$$\texttt{NoAssignedRefTo(e)} = \{\,\texttt{a},\texttt{b}\,\} \tag{3.37}$$

If e is assigned fprivate of δ, then declarations a and b will also get assigned to δ.



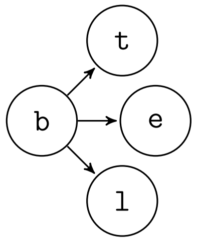
$$\begin{aligned}\texttt{field(e)} \wedge \texttt{F(e)} &= \perp\\ \texttt{field(a)} \wedge \texttt{F(a)} &= \perp\\ \texttt{method(b)} \wedge \texttt{F(b)} &= \perp\\ \texttt{field(c)} \wedge \texttt{F(c)} &= \alpha\end{aligned}$$

Figure 3.12: Inference 1. fprivate Propagation

Note: If there are declarations in RefTo(e) that have been already assigned to a feature, this is considered in possibilities calculation for e. B^t_tF does not allow feature assignment overriding. Feature bounds and possibilities for e prevents a user from choosing invalid options.

Caveats

Hooks that reference fprivate declarations.
IN Figure 3.12, consider method b. Figure 3.13 shows the CRG corresponding to b. Let say that b is a hook, and it does not belong to feature δ, however it was assigned to it by inference.



$$\begin{aligned}\texttt{method(b)} \land \texttt{F(b)} = \delta \land \texttt{parent(b)} = \texttt{t}\\ \texttt{container(t)} \land \texttt{F(t)} = \beta\\ \texttt{field(e)} \land \texttt{F(e)} = \delta\\ \texttt{field(l)} \land \texttt{F(l)} = \bot\end{aligned}$$

Figure 3.13: Inference 1. `fprivate` Propagation Caveats

For a method that has been assigned by inference, like in the case of `b`, `BttF` determines whether to keep that assignment or ask a user for `b`'s assignment. If `b` has other references besides the one to `a` and the one to its parent, `t`, then `BttF` would ask for `b` assignment. It this case `b` does have another reference to field `f`, therefore `BttF` will remove `b`'s current assignment and ask a user about `b`'s assignment. If there was no other reference, then `b` would remain assigned to δ. If a user still considers that `b` should not be assigned to δ, then `e` cannot be assigned `fprivate`, it would have to be `fpublic` so it does not affect `b`'s assignment. The rule for determining when to remove a method's by inference assignment is as follows:

$$\frac{\begin{array}{c}\texttt{method(b)} \land \texttt{F(b)} \neq \bot \land \texttt{ByInference(b)}\\ \land \big(\texttt{RefFrom(b)} - \{\texttt{parent(b)} \cup \texttt{ByInferenceDec(b)}\}\big) \neq \emptyset\end{array}}{\texttt{F(b)} = \bot} \tag{3.38}$$

3.4.2 Inference 2: `fprivate` Containers

If `e` is an `fprivate` container of feature `F`, declarations in `NoAssignedRefTo(e)` that are not `e`'s children are automatically assigned to `F` without a modifier assignment. And declarations in `NoAssignedRefTo(e)` that are `e`'s children are also automatically assigned to `F` but with `fprivate` modifier to continue the assignments recursively.

Let's partition `NoAssignedRefTo(e)` into `ChildOf(e)` and `NoChildOf(e)`. `ChildOf(e)` contains those declarations in `NoAssignedRefTo(e)` that have `e` as parent.[25] `NoChildOf(e)` contains those declarations in `NoAssignedRefTo(e)` that do not have `e` as parent, this means they reference `e` but are not contained in `e`.

$$\begin{aligned}\texttt{NoAssignedRefTo(e)} &= \texttt{ChildOf(e)} \cup \texttt{NoChild(e)} &(3.39)\\ \texttt{ChildOf(e)} &= \{\texttt{r} \mid \texttt{r} \in \texttt{NoAssignedRefTo(e)} \land \texttt{parent(r)} = \texttt{e}\} &(3.40)\\ \texttt{NoChildOf(e)} &= \{\texttt{r} \mid \texttt{r} \in \texttt{NoAssignedRefTo(e)} \land \texttt{parent(r)} \neq \texttt{e}\} &(3.41)\end{aligned}$$

Declarations in `ChildOf(e)` will be assigned `fprivate` of `F`, this makes that all levels contained in `e` get also assigned `fprivate` of `F`.[26] Declarations in `NoChildOf(e)` will also be

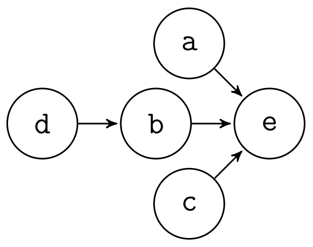
[25] A parent of a declaration is its container.
[26] Assignment of a container's child happens recursively.

assigned to F, but without modifier.

$$\frac{\texttt{container(e)} \land \texttt{fprivate(e)} \land \texttt{F(e)} = \texttt{F}}{\begin{array}{c}\forall \texttt{r}\left(\texttt{r} \in \texttt{ChildOf(e)} \Rightarrow \left((\texttt{F(r)} = \texttt{F}) \land \texttt{fprivate(r)}\right)\right) \\ \land \forall \texttt{q}\left(\texttt{q} \in \texttt{NoChildOf(e)} \Rightarrow (\texttt{F(r)} = \texttt{F})\right)\end{array}} \tag{3.42}$$

Note: Consider CRG in Figure 4.9. Suppose e is a container. Declarations a and c reference e but are not contained in e. Declaration b is a container that is contained in e. Declaration d is contained in b. We have:

$$\begin{aligned}\texttt{NoAssignedRefTo(e)} &= \{\,\texttt{a,b,c}\,\} \\ \texttt{NoAssignedRefTo(b)} &= \{\,\texttt{d}\,\}\end{aligned}$$



$$\begin{aligned}\texttt{container(e)} \land \texttt{F(e)} &= \perp \\ \texttt{method(a)} \land \texttt{parent(a)} \neq \texttt{e} \land \texttt{F(a)} &= \perp \\ \texttt{container(b)} \land \texttt{parent(b)} = \texttt{e} \land \texttt{F(b)} &= \perp \\ \texttt{field(c)} \land \texttt{parent(c)} \neq \texttt{e} \land \texttt{F(c)} &= \perp \\ \texttt{method(d)} \land \texttt{parent(d)} = \texttt{b} \land \texttt{F(d)} &= \perp\end{aligned}$$

Figure 3.14: Inference 2. `fprivate` Container

If we divide NoAssignedRefTo(e) and NoAssignedRefTo(b) into their corresponding ChildOf() and NoChildOf() subsets, we have:

$$\begin{aligned}\texttt{ChildOf(e)} &= \{\,\texttt{b}\,\} \\ \texttt{NoChildOf(e)} &= \{\,\texttt{a,c}\,\} \\ \texttt{ChildOf(b)} &= \{\,\texttt{d}\,\} \\ \texttt{NoChildOf(b)} &= \emptyset\end{aligned}$$

Suppose container e is assigned `fprivate` of γ. Then declarations a and c will be assigned to γ without modifier. Declaration b will be assigned `fprivate` of γ, given this d will also get assigned recursively `fprivate` of γ.

Caveats

Hooks that reference `fprivate` declarations.
As with Inference 1, for a method in NoChildOf(e) assigned by inference, like in the case of declaration a, B$^{t}_{t}$F determines whether to keep that assignment or ask a user for a's assignment. As a reader may notice this applies only to methods in NoChildOf(e). The inferences made on ChildOf(e) are not undone by B$^{t}_{t}$F. For method a, if it has other references besides the one to e and the one to its parent, then B$^{t}_{t}$F will remove its current assignment

and ask a user about a's assignment. Otherwise, current a's assignment will remain. The rule for determining when to remove a method's by inference assignment in this case is as follows:

$$\frac{\begin{array}{c}\mathtt{method(a)} \wedge \mathtt{F(a)} \neq \perp \wedge \mathtt{ByInference(a)} \wedge \big(\mathtt{parent(a)} \neq \mathtt{ByInferenceDec(a)}\big) \\ \wedge \big(\mathtt{RefFrom(a)} - \{\mathtt{parent(a)} \cup \mathtt{ByInferenceDec(a)}\}\big) \neq \emptyset\end{array}}{\mathtt{F(a)} = \perp} \tag{3.43}$$

Note: Equation (3.43) is comprehensive of Equation (3.42) in Inference 1 caveats. Equation (3.42) occurs in an scenario where condition `parent(a)` ≠ `ByInferenceDec(a)` is always true. Equation (3.42) is applied when inferences are made based on a `fprivate` field or method, which for B$^{\mathrm{t}}_{\mathrm{t}}$F's purposes are never containers, therefore neither parents.

Hook that is a child of a `fprivate` container.
Let say we have a method `m` that was previously identified as hook but its assignment was removed because of caveats in Inference 1 or Inference 2, and now its container is assigned `fprivate` of a feature `F`. In this case, method `m` will be assigned to feature `F` too, but its modifier will remain `fpublic`, hooks are always `fpublic`.

3.4.3 Inference 3: Declarations in latest feature are always `fprivate`

References from declarations across features flow from ω, the latest layer, to α, the base layer. This means that a declaration references other declarations in the same layer or in earlier layers. The only exception are hooks, that could reference declarations in the future.

The CRG in Figure 3.15 shows how references flow from ω to α. Declarations in ω will not be referenced by other declarations except for hook methods in earlier features, an example of the latter is the edge `d` → `i` where `d` is a hook. Making all declarations in ω `fprivate`, increases the number of inferences, and does not affect declarations visibility, declarations in ω can be referenced by hooks, because they can reference `fprivate` declarations in the future.

The rule for a declaration `e` assigned to ω is:

$$\frac{\mathtt{F(e)} = \omega}{\mathtt{fprivate(e)}} \tag{3.44}$$

For recursive partitioning this rule only applies to the latest sub-feature of ω, the latest feature. Consider recursive feature model in Equation (3.45), for this feature model, declarations assigned to feature `Y` will always be `fprivate`. Declarations assigned to `Y` are not referenced by declarations in other features, except for hooks, so their visibility is not

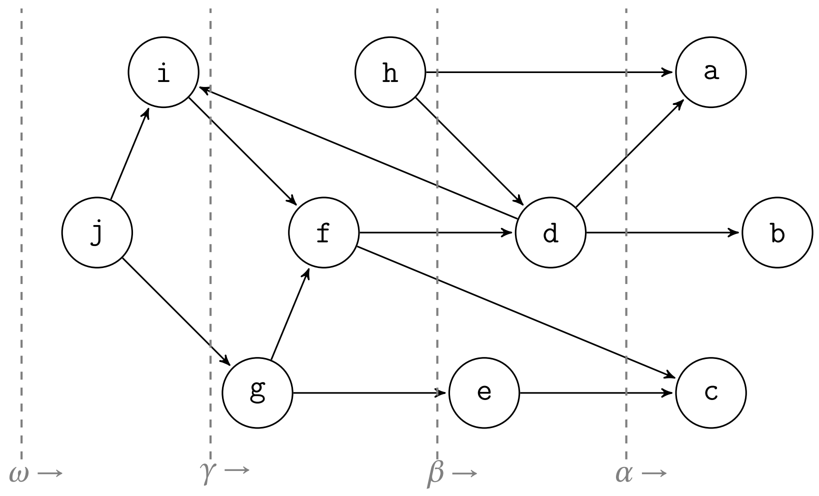


Figure 3.15: Inference 3. References flow across features

affected by a `fprivate` modifier, and yet the amount of BttF's inferences increases, decreasing a user's input.

```
P : A B C
A : R S
C : X Y
```
(3.45)

3.4.4 Inference 4: One Feature in Bounds

If FB(e) contains a single feature, BttF will assign e to it. In this case a user will not be asked for a modifier. As mentioned before, BttF's goal is to get feature assignments, and in this case, a feature assignment can be done without a modifier. Deciding which modifier to apply to a declaration is a sensitive user task, that could affect a declaration's visibility. This decision should be based on a user's knowledge of the program.

Considering that `fprivate` modifier is BttF's main resource for making inferences, it could be argued that not asking for a modifier might be a loss of information. But, consider CRG in Figure 3.16. Suppose FB(a) = $[\delta,\delta]$. Further assume that b is not a method[27] and FB(b) will have more than one feature.

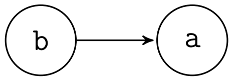


Figure 3.16: Simple CRG.

There are two ways to approach this:

(1) Do not ask for a's modifier. This is BttF's current approach. The amount of user inputs required in this case is 1. This can be observed in Table 3.9.

[27] As can be observed in previous inferences, methods do not always get assigned by inference.

No.	**Step**	**User's input count**
1	B$^{t}_{t}$F infers F(a) = δ	0
2	B$^{t}_{t}$F asks for b assignment.	1
	Total:	1

Table 3.9: Do not ask for a's modifier.

(2) Ask for a's modifier. There are two possible cases here. A user says that a is `fprivate`. As shown in Table 3.10, the amount of user inputs required in this case is 2. The other possible case is that a user says that a is `fpublic`. As shown in Table 3.11, the amount of user inputs required in this case is also 2.

No.	**Step**	**User's input count**
1	B$^{t}_{t}$F infers F(a) = δ.	0
2	B$^{t}_{t}$F asks for a's modifier. User says `fprivate`(a).	1
3	B$^{t}_{t}$F assigns F(b) = δ by inference.	0
4	B$^{t}_{t}$F asks b's modifier.	1
	Total:	2

Table 3.10: Ask for a's modifier. User says `fprivate`(a).

No.	**Step**	**User's input count**
1	B$^{t}_{t}$F infers F(a) = δ.	0
2	B$^{t}_{t}$F asks for a's modifier. User says `fpublic`(a).	1
3	B$^{t}_{t}$F asks for b's assignment (feature-modifier assignment).	1
	Total:	2

Table 3.11: Ask for a's modifier. User says `fpublic`(a).

The comparison of both approaches shows that B$^{t}_{t}$F's current approach is aligned to its goal and it also requires less user inputs than the alternative. If a user insists on assigning modifiers in such cases, it is possible via a declarations assignment file. Details about this are in Chapter 4.

3.4.5 Inference 5: Declaration with no References

This inference applies for a declaration e that is not referenced by other declarations, and that only references its container. In this case e is automatically assigned to the same feature of its container.

$$\frac{\mathtt{RefTo(e)} = \emptyset \ \wedge \ \mathtt{RefFrom(e)} = \mathtt{parent(e)}}{\mathtt{F(e)} = \mathtt{F(parent(e))}} \tag{3.46}$$

`RefTo(e)` = $\emptyset$ means that e is not referenced by other declarations and neither contains other declarations. And if it only references it container then e is dead code. In this case e's assignment will not affect other declarations assignment.

3.5 Recursive Partitioning

B^t_tF supports the recursive partitioning of a program. By *recursive* we mean partitioning a program into a set of features, and then sub-partitioning at least one of the original features into sub-features. This process can be repeated any number of times. The following subsections explain how recursive partitioning in B^t_tF works.

3.5.1 Processing of a Feature Model for Recursive Partitioning

A B^t_tF's feature model for recursive partitioning has the form of a feature model in Equation (3.47). `P` represents the program to partition. `A`, `B` and `C` are its initial features where `A` < `B` < `C`. `B` is partitioned into sub-features `M` < `N`. `C` is partitioned into sub-features `X` < `Y`.

$$\begin{array}{l}\texttt{P : A B C}\\ \texttt{B : M N}\\ \texttt{C : X Y}\end{array} \tag{3.47}$$

B^t_tF requires a user to partition a program in an ordered manner: from less granular features to more granular. Features `B` and `C` need to have declarations assigned to them before they can be sub-partitioned. Since `B` and `C` have the same granularity, either of them can be partitioned in any order; the order is specified by the B^t_tF feature model. Given this, there are two valid forms to process this feature model partitioning:

Option 1:

1. `P : A B C`
2. `B : M N`
3. `C : X Y`

Or, Option 2:

1. `P : A B C`
2. `C : X Y`
3. `B : M N`

A user first has to partition completely program `P` into features `A`, `B` and `C`, then s/he can partition `B` or `C`.

B^t_tF works on one line of a feature model at a time. A user picks which line of a feature model to process next. B^t_tF verifies that a picked line is valid for processing. A line can be processed if all the following are true:

1. The parent of a feature to sub-partition has been fully partitioned. For example, feature B cannot be sub-partitioned if there are declarations in P that have not yet been assigned to either A, B or C.
2. The feature to sub-partition exists. For example, it is considered that B exists if there are declarations assigned to it.

If all lines of feature model in Equation (3.47) are processed completely, declarations in program P will be assigned to one of these features A, M, N, X or Y. At the end there are no declarations assigned to B or C.

3.5.2 Reassigning Declarations to Sub-Features

For the first line of a recursive partitioning feature model, as for a non-recursive partitioning feature model, a user has to assign all declarations in a program's CRG, this set of declarations is contained in PD. At the initial state, any declaration in PD have been assigned to a feature:

$$\forall d\,(\,d \in PD \Rightarrow F(d) = \bot\,) \tag{3.48}$$

After a user has assigned all declarations according to the first line of a feature model, we have:

$$\forall d\,(\,d \in PD \Rightarrow F(d) \neq \bot\,) \tag{3.49}$$

This holds true for the rest of the partitioning process when there is a recursive feature model. The process of sub-partitioning a feature G consists of reassigning its declarations to the sub-features of G, say X < Y. On this scenario B$^{t}_{t}$F creates a view of the declarations to reassign PD′:

$$PD' = \{\,d \in PD \wedge F(d) = G\,\} \tag{3.50}$$

After a user has finished reassigning declarations of G to its sub-features:

$$\forall d\big(\,d \in PD \wedge F(d) \neq G\,\big) \bigwedge \forall d\big(\,d \in PD' \Rightarrow F(d) = X \vee F(d) = Y\,\big) \tag{3.51}$$

B$^{t}_{t}$F's Rule Variants for Recursive Partitioning

B$^{t}_{t}$F applies the same rules for declarations in both PD and PD′, except for possibilities calculation. B$^{t}_{t}$F calculates new feature bounds for a declaration e to be reassigned to a sub-feature of G. But it *does not* recalculate modifiers. The current modifier of e remains, only its feature assignments change. In this way no information is lost.

Let's say that G is to be partitioned into sub-features X and Y:

$$\texttt{G : X Y}$$

And that e is fprivate of G:

$$(\texttt{F(e)} = \texttt{G}) \wedge \texttt{fprivate(e)}$$

BttF will calculate which features from X and Y are in bounds for e. If both features are in bounds, then a user will have these possibilities:

$$\mathbb{P}(\texttt{X,e,fprivate})$$
$$\mathbb{P}(\texttt{Y,e,fprivate})$$

If only X or Y are in bounds, then BttF will infer e's assignment keeping e's fprivate modifier.

Caveats
If e's fprivate modifier is not correct, a user may have to retract this fact in a previous step of the feature model process. Let's say that our feature model is:

```
P : B G T
G : X Y
```

Suppose e was assigned fprivate of G when a user processed the first line of this feature model. While processing the second line, a user may realize that e should be fpublic. In this case, s/he has to go back and select the first line of this feature model for processing, and retract the fact where it was specified that e was fprivate of G, and provide the correct assignment. This, in turn, may trigger feature assignments to other declarations.

BttF allows a user to save her/his work done in a CSV file. In this scenario if a user needs to retract facts in a previous step, s/he can save her/his work (which is saved with the current feature granularity level), go and modify the assignments done in a previous step, come back and upload the assignments file s/he saved. All the assignments that remain valid are applied, the ones that do not, are notified. More details about uploading an assignments CSV file are discussed in next chapter.

Chapter 4

BttF Execution

BttF's can be executed in two modes: As an Eclipse[1] plugin (see Figure 4.1) and standalone[2]. The Eclipse plugin version is the fullest version of BttF. However, it is available only for Eclipse Java programs. Table 4.1 shows the differences between both execution modes.

	Plugin	**Standalone**
Program Type	Only Eclipse Java programs.	Any program.
CRG	BttF automatically obtains a program's CRG using Eclipse's AST framework [31].	User needs to obtain a program's CRG by other means and upload it to BttF.
Output	BttF provides a user with a `.csv` containing the list of feature assignments. Additionally, for SPL program partitioning, BttF makes feature annotations in the source code.	BttF provides a user with a `.csv` containing the list of feature assignments. The source code is not modified by BttF.

Table 4.1: Differences between BttF's Execution Modes

4.1 BttF's Input

After a program's CRG has been obtained, a user may provide a feature model file, and a list of declarations' assignments in a `.csv` file. BttF asks for a state folder as input, which should contain a `.bttf` file that is a feature model in BttF's accepted format, and a declaration assignment `.csv` file (`BttF.csv`). If a user does not provide a valid state folder, he or she will have to manually input a feature model in BttF's user interface.

[1] Eclipse Neon Version is supported.

[2] Executable jar file.

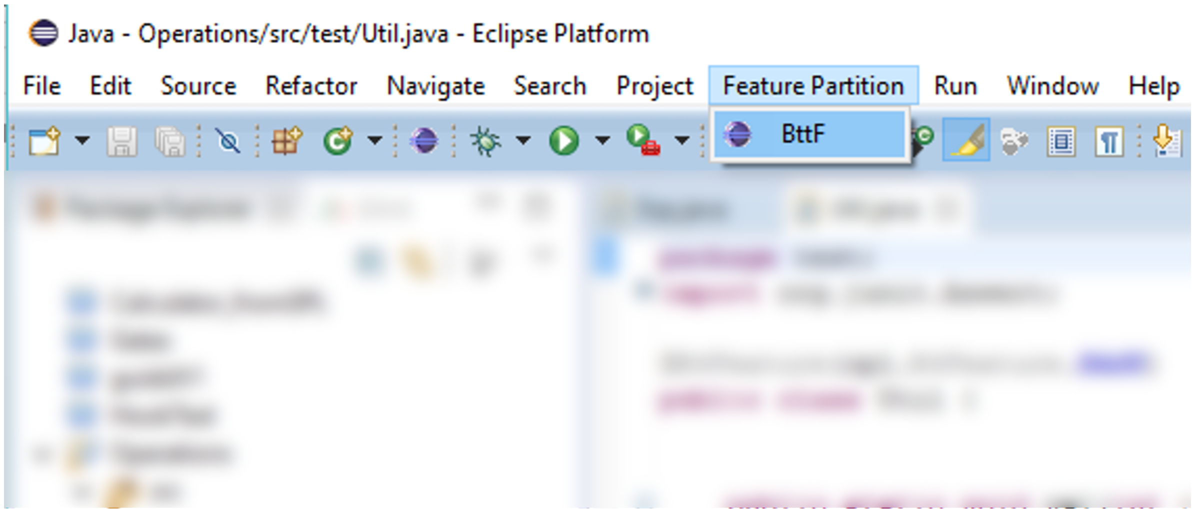
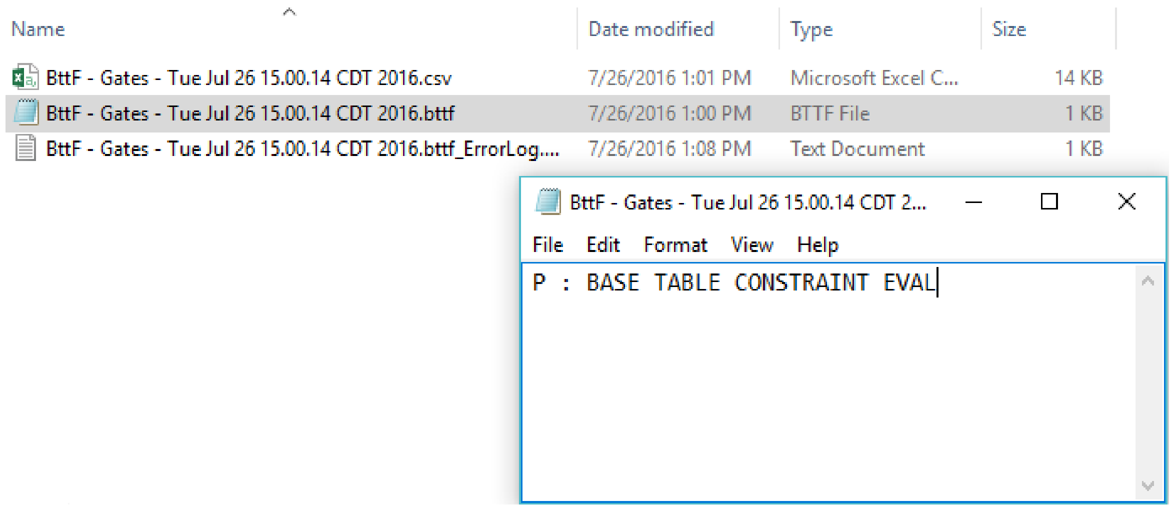


Figure 4.1: BᵗₜF as Eclipse's plugin

An example of a `.bttf` file is shown in Figure 4.2. The feature model contained in this file specifies four features: `BASE`, `TABLE`, `CONSTRAINT` and `EVAL`.



Name	Date modified	Type	Size
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.csv	7/26/2016 1:01 PM	Microsoft Excel C...	14 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf	7/26/2016 1:00 PM	BTTF File	1 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf_ErrorLog....	7/26/2016 1:08 PM	Text Document	1 KB

BttF - Gates - Tue Jul 26 15.00.14 CDT 2...
File Edit Format View Help

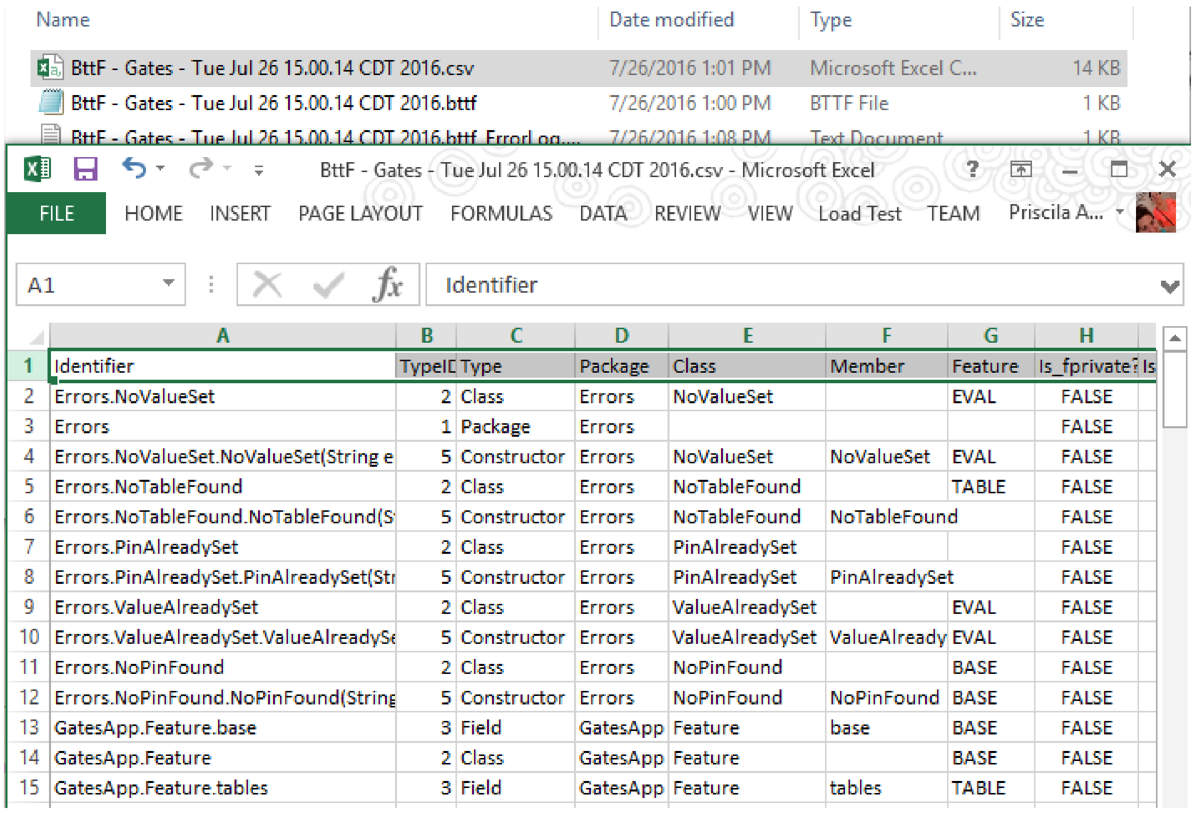
```
P : BASE TABLE CONSTRAINT EVAL
```

Figure 4.2: A `.bttf` file.

After a feature model file has been provided manually, BᵗₜF gives a user the option to upload a `BttF.csv` file. A user also may upload a `BttF.csv` file later, after he or she has started making feature assignments using BᵗₜF user interface.

Figure 4.3 shows an example of a `BttF.csv` file. This file allows a user to provide feature and modifier information about declarations. A user does not have to provide information about all declarations. To get a `.csv` file with the appropriate format for BᵗₜF to read it, a user can download it from BᵗₜF's user interface.

When a user uploads a `BttF.csv` file, BᵗₜF processes it in the following way:



Name	Date modified	Type	Size
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.csv	7/26/2016 1:01 PM	Microsoft Excel C...	14 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf	7/26/2016 1:00 PM	BTTF File	1 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf_ErrorLog....	7/26/2016 1:08 PM	Text Document	1 KB

BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.csv - Microsoft Excel

A1 Identifier

	A	B	C	D	E	F	G	H
1	Identifier	TypeID	Type	Package	Class	Member	Feature	Is_fprivate? Is
2	Errors.NoValueSet	2	Class	Errors	NoValueSet		EVAL	FALSE
3	Errors	1	Package	Errors				FALSE
4	Errors.NoValueSet.NoValueSet(String e	5	Constructor	Errors	NoValueSet	NoValueSet	EVAL	FALSE
5	Errors.NoTableFound	2	Class	Errors	NoTableFound		TABLE	FALSE
6	Errors.NoTableFound.NoTableFound(S	5	Constructor	Errors	NoTableFound	NoTableFound		FALSE
7	Errors.PinAlreadySet	2	Class	Errors	PinAlreadySet			FALSE
8	Errors.PinAlreadySet.PinAlreadySet(Str	5	Constructor	Errors	PinAlreadySet	PinAlreadySet		FALSE
9	Errors.ValueAlreadySet	2	Class	Errors	ValueAlreadySet		EVAL	FALSE
10	Errors.ValueAlreadySet.ValueAlreadySe	5	Constructor	Errors	ValueAlreadySet	ValueAlready	EVAL	FALSE
11	Errors.NoPinFound	2	Class	Errors	NoPinFound		BASE	FALSE
12	Errors.NoPinFound.NoPinFound(String	5	Constructor	Errors	NoPinFound	NoPinFound	BASE	FALSE
13	GatesApp.Feature.base	3	Field	GatesApp	Feature	base	BASE	FALSE
14	GatesApp.Feature	2	Class	GatesApp	Feature		BASE	FALSE
15	GatesApp.Feature.tables	3	Field	GatesApp	Feature	tables	TABLE	FALSE

Figure 4.3: A BttFcsv file.

1. **Content structure check**. BttF verifies that the contents of the file are structurally valid. The expected columns should be present and the declaration identifier should match with columns package, class and member.[3] If the file does not pass this verification, the user is notified and the file's contents are not read.

2. **Content validation**. BttF verifies that every declaration with a feature assignment has a valid identifier, a valid declaration type and a valid feature name. If one of those are not valid, this declaration assignment is not considered. When a declaration fails this validation, it is added to the error log file with the corresponding reason.

   (a) **Feature validation when recursive partitioning**. If a program is to be partitioned recursively, and a feature assignment has a feature F that is not found in the current line of a feature model, then BttF searches feature F containers until it finds a valid one. If a valid feature cannot be found, this is added to the error log file. As an example, consider feature model in Equation (4.1). Suppose a user partitions program P, so the line to use is the first one of this feature model. If a user provides a feature assignment with feature X, BttF will search

[3] The identifier is present as a single column, and also decomposed in three columns: package, class and member. This for easing users' sorting and filtering activities.

across the feature model and find the corresponding mapping for `X`, which is `A`.

$$\begin{aligned}&\mathtt{P : A\ B\ C}\\&\mathtt{A : S\ T}\\&\mathtt{S : X\ Y}\end{aligned} \tag{4.1}$$

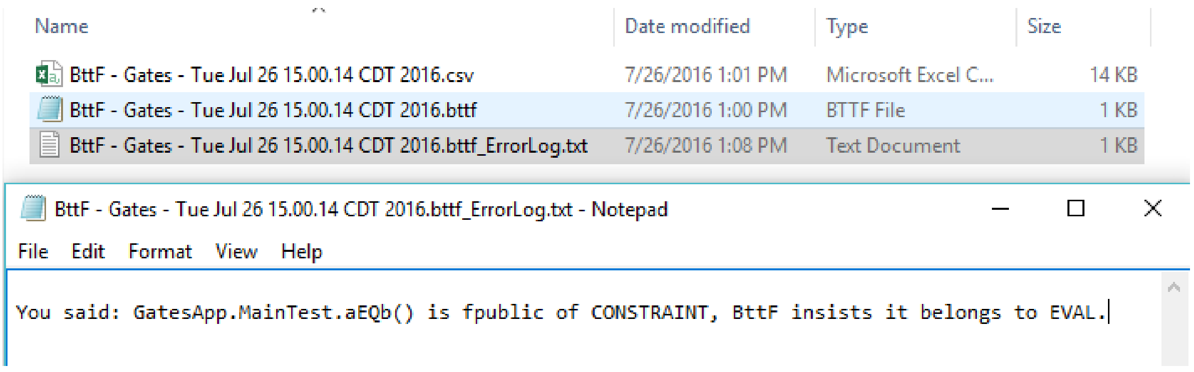
3. **Declarations sorting**. Declarations that passed previous steps are sorted on granularity, from packages to fields.

4. **Feature assignment**. Two possible scenarios may happen at this point:

   (a) **Declaration has been already assigned**. A declaration may get assigned because it references a `fprivate` declaration, or because it already has been assigned using BttF user interface. If a user provided assignment is different from the existing one, the existing one *is not overridden* and this is added to the error log file. This means that BttF still asserts that the original feature, not the newly proposed one, is correct.

   (b) **Declaration has not been assigned**. Feature bounds are calculated for this declaration. If a user assigned feature is in bounds the feature-modifier assignment is applied. In case of a `fprivate` assignment, the corresponding propagation is done. If a user assignment is out of bounds then this is added to the error log file.

5. **Save error log file**. All found errors are outputted in a file and notified to the user. Figure 4.4 shows an example of a BttF's error log file. This file is notifying a user that a feature assignment is not valid, and the actual feature assignment that BttF made with reference to that declaration.



Name	Date modified	Type	Size
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.csv	7/26/2016 1:01 PM	Microsoft Excel C...	14 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf	7/26/2016 1:00 PM	BTTF File	1 KB
BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf_ErrorLog.txt	7/26/2016 1:08 PM	Text Document	1 KB

BttF - Gates - Tue Jul 26 15.00.14 CDT 2016.bttf_ErrorLog.txt - Notepad
File
Edit
Format
View
Help

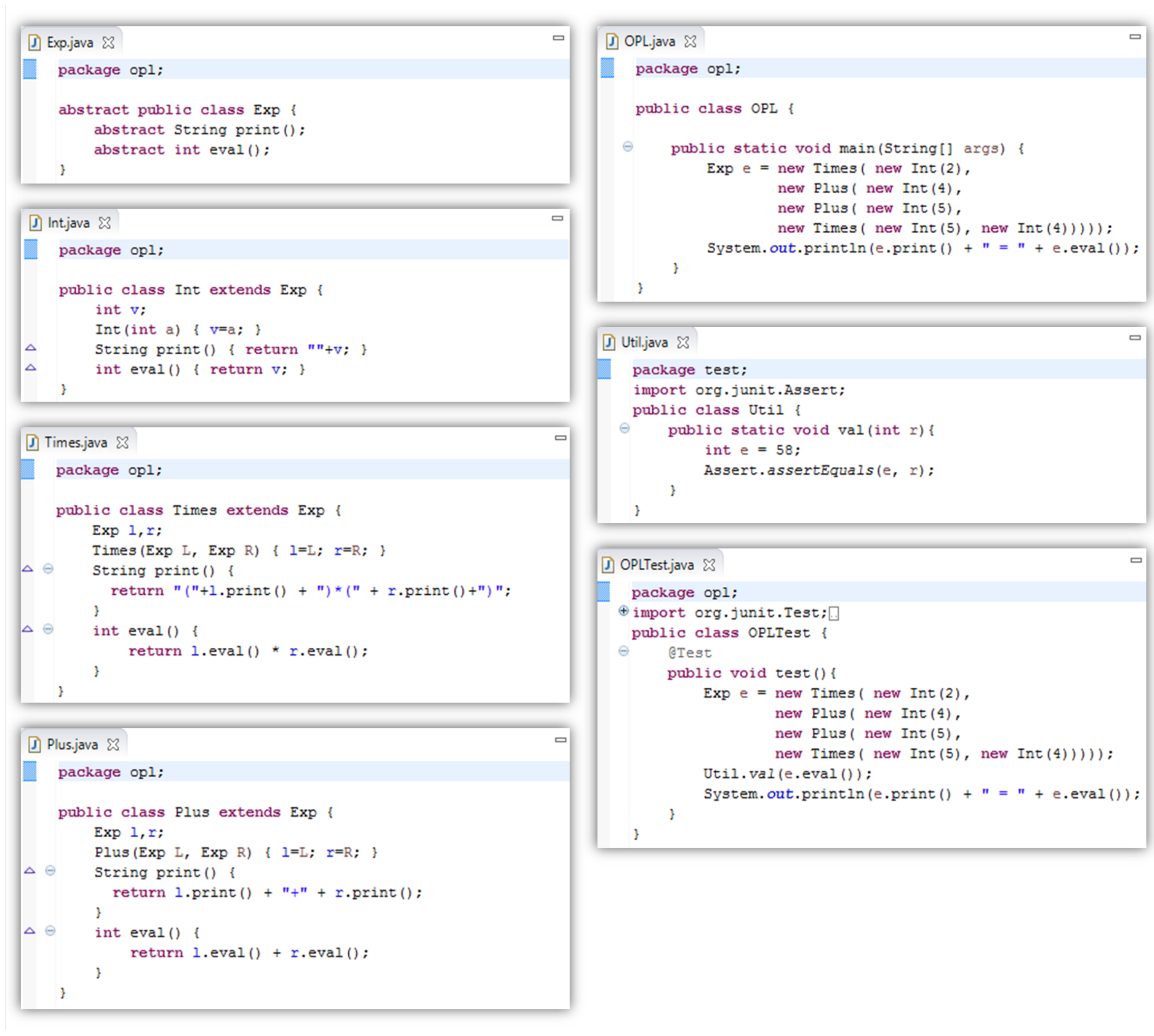
```
You said: GatesApp.MainTest.aEQb() is fpublic of CONSTRAINT, BttF insists it belongs to EVAL.
```

Figure 4.4: A BttF's error log file.

6. **Continue on BttF's user interface**. A user may continue with pending feature assignments in BttF's user interface.

4.2 BttF’s User Interface

For this section and the following one, we use as an example, Java program `Operations`. Figure 4.5 shows the source code of `Operations`. This program implements times and plus operations for integer values. Its CRG has 28 vertices (declarations) and 85 edges. The program structure can be observed in Figure 4.6.



Exp.java

```java
package opl;

abstract public class Exp {
    abstract String print();
    abstract int eval();
}
```

Int.java

```java
package opl;

public class Int extends Exp {
    int v;
    Int(int a) { v=a; }
    String print() { return ""+v; }
    int eval() { return v; }
}
```

Times.java

```java
package opl;

public class Times extends Exp {
    Exp l,r;
    Times(Exp L, Exp R) { l=L; r=R; }
    String print() {
      return "("+l.print() + ")*(" + r.print()+")";
    }
    int eval() {
        return l.eval() * r.eval();
    }
}
```

Plus.java

```java
package opl;

public class Plus extends Exp {
    Exp l,r;
    Plus(Exp L, Exp R) { l=L; r=R; }
    String print() {
      return l.print() + "+" + r.print();
    }
    int eval() {
        return l.eval() + r.eval();
    }
}
```

OPL.java

```java
package opl;

public class OPL {

    public static void main(String[] args) {
        Exp e = new Times( new Int(2),
                  new Plus( new Int(4),
                  new Plus( new Int(5),
                  new Times( new Int(5), new Int(4)))));
        System.out.println(e.print() + " = " + e.eval());
    }
}
```

Util.java

```java
package test;
import org.junit.Assert;
public class Util {
    public static void val(int r){
        int e = 58;
        Assert.assertEquals(e, r);
    }
}
```

OPLTest.java

```java
package opl;
import org.junit.Test;
public class OPLTest {
    @Test
    public void test(){
        Exp e = new Times( new Int(2),
                  new Plus( new Int(4),
                  new Plus( new Int(5),
                  new Times( new Int(5), new Int(4)))));
        Util.val(e.eval());
        System.out.println(e.print() + " = " + e.eval());
    }
}
```

Figure 4.5: Operations Java Program Source Code

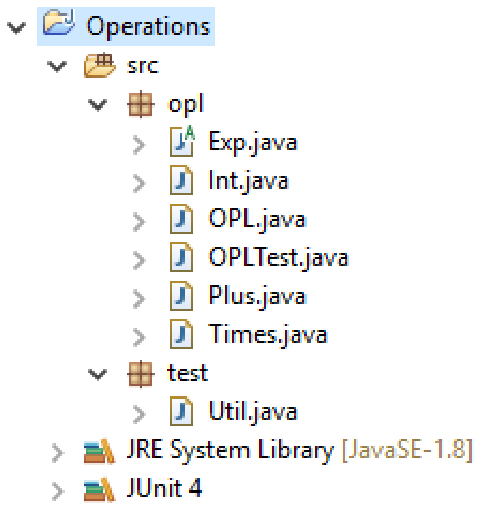
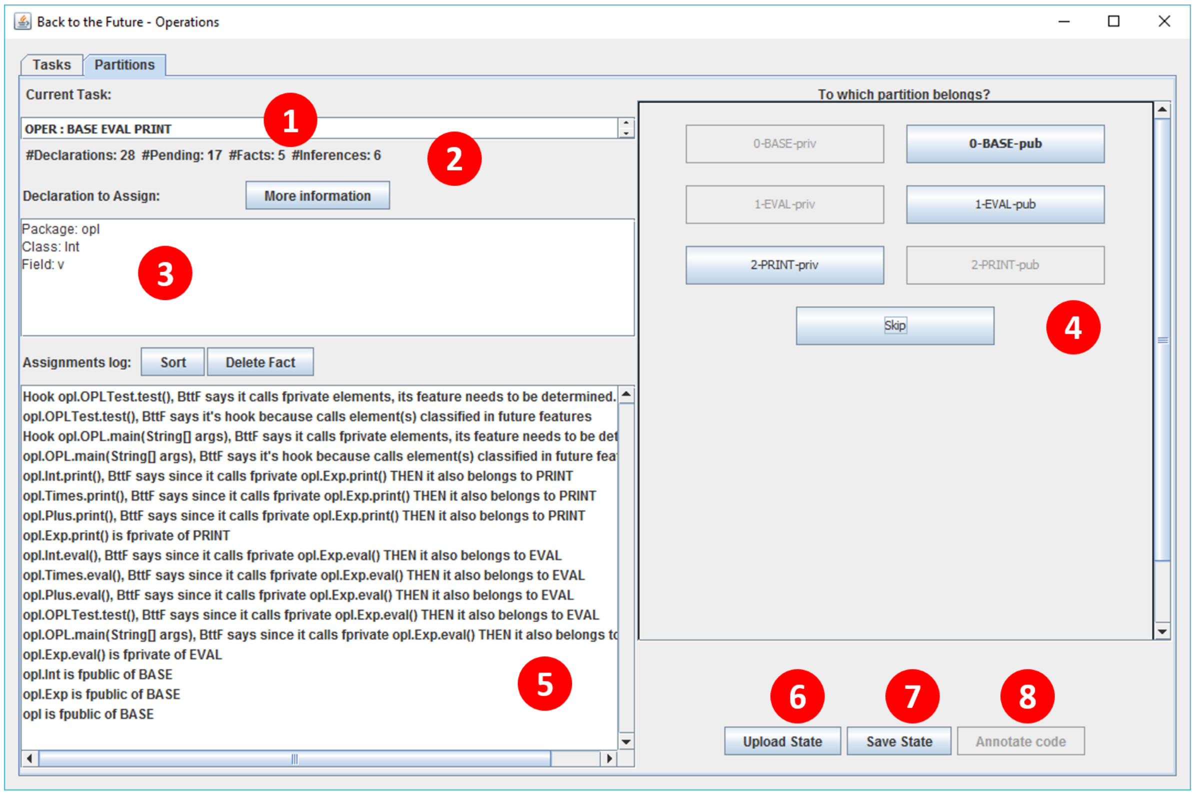


Figure 4.6: Operations Java Program

`Operations` was partitioned using B$^{t}_{t}$F' Eclipse plugin version. Figure 4.7 shows a B$^{t}_{t}$F's screen-shot taken while partitioning `Operations` program. The elements in B$^{t}_{t}$F's user interface are explained next.



```
OPER : BASE EVAL PRINT
#Declarations: 28  #Pending: 17  #Facts: 5  #Inferences: 6
```

```
Package: opl
Class: Int
Field: v
```

```
Hook opl.OPLTest.test(), BttF says it calls fprivate elements, its feature needs to be determined.
opl.OPLTest.test(), BttF says it's hook because calls element(s) classified in future features
Hook opl.OPL.main(String[] args), BttF says it calls fprivate elements, its feature needs to be det
opl.OPL.main(String[] args), BttF says it's hook because calls element(s) classified in future fea
opl.Int.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
opl.Times.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
opl.Plus.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
opl.Exp.print() is fprivate of PRINT
opl.Int.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
opl.Times.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
opl.Plus.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
opl.OPLTest.test(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
opl.OPL.main(String[] args), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to
opl.Exp.eval() is fprivate of EVAL
opl.Int is fpublic of BASE
opl.Exp is fpublic of BASE
opl is fpublic of BASE
```

Figure 4.7: Partition Operations Program

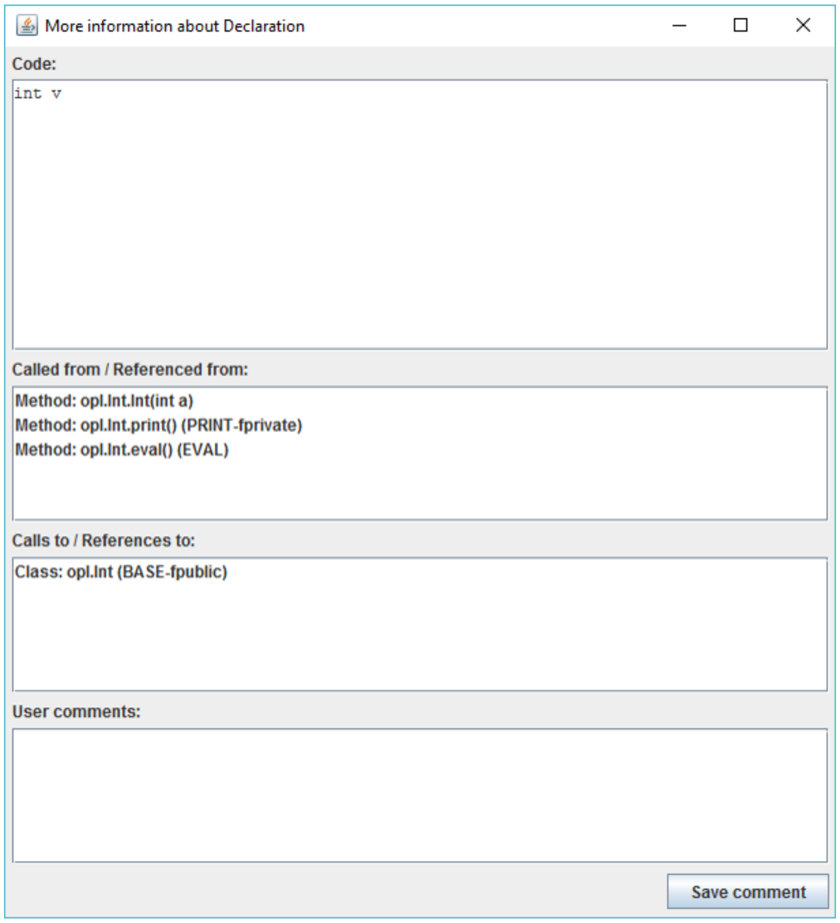
- **Bubble 1** indicates the current partitioning task, which in this case is the only line of

Operations' feature model:

$$\texttt{OPER : BASE EVAL PRINT} \tag{4.2}$$

- **Bubble 2** shows the current partitioning status: of a total of 28 declarations, 17 are still pending to assign, a user has provided 5 facts or assignments, and BttF has made 6 inferences so far.

- **Bubble 3** shows the current declaration to assign, it this case is field `v` from class/type `int`. The `'More information'` button shows a screen that contains current declaration's code, the list of declarations that reference it, and the list of declarations it references, it also includes an editable field where a user may input a comment regarding to the current declaration. This screen can be observed in Figure 4.8.



More information about Declaration

Code:

```
int v
```

Called from / Referenced from:

Method: opl.Int.Int(int a)
Method: opl.Int.print() (PRINT-fprivate)
Method: opl.Int.eval() (EVAL)

Calls to / References to:

Class: opl.Int (BASE-fpublic)

User comments:

Save comment

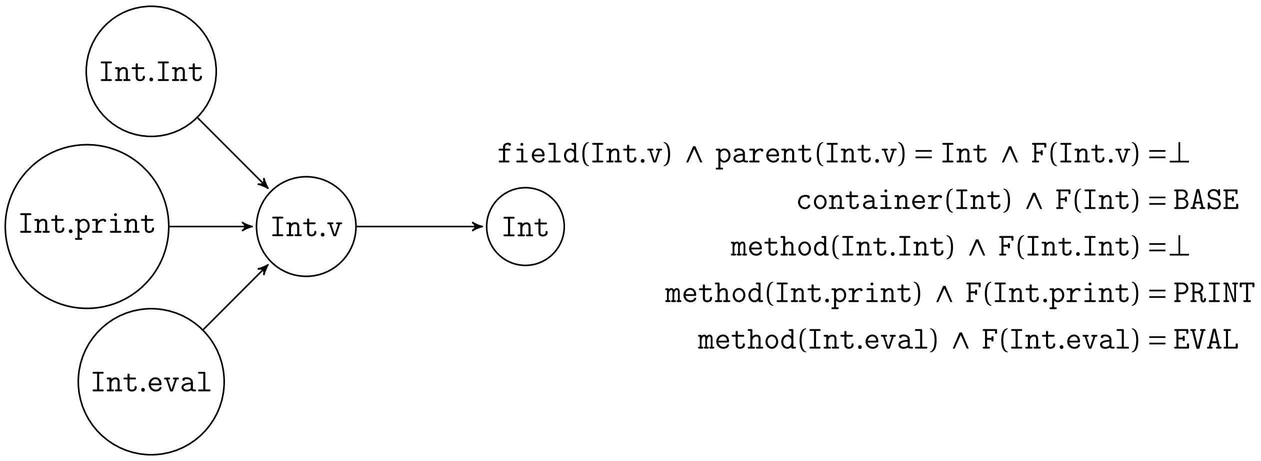
Figure 4.8: BttF's More Information about a Declaration.

- **Bubble 4** points to the set of possibilities for field `v`. Figure 4.9 shows field `v`'s CRG. `v`'s earliest bound is `BASE`, and its latest bound is `PRINT`. Therefore, `v`'s bounds are:

$$\texttt{FB(v)} = \{\texttt{BASE, EVAL, PRINT}\} \tag{4.3}$$

v can be fpublic of any feature in bounds, except for PRINT which is the latest feature, therefore all of its declarations are always fprivate.[4] None of the fprivate possibilities cases apply for v. v's possibilities are:

$$\mathbb{P}(\mathtt{BASE},\mathtt{v},\mathtt{fpublic})$$
$$\mathbb{P}(\mathtt{EVAL},\mathtt{v},\mathtt{fpublic})$$
$$\mathbb{P}(\mathtt{PRINT},\mathtt{v},\mathtt{fprivate})$$



$$\mathtt{field(Int.v)} \land \mathtt{parent(Int.v)} = \mathtt{Int} \land \mathtt{F(Int.v)} = \perp$$
$$\mathtt{container(Int)} \land \mathtt{F(Int)} = \mathtt{BASE}$$
$$\mathtt{method(Int.Int)} \land \mathtt{F(Int.Int)} = \perp$$
$$\mathtt{method(Int.print)} \land \mathtt{F(Int.print)} = \mathtt{PRINT}$$
$$\mathtt{method(Int.eval)} \land \mathtt{F(Int.eval)} = \mathtt{EVAL}$$

Figure 4.9: Field Int.v's CRG

- **Bubble 5** points to the 'Assignments log', which contains the list of facts that a user has provided and the inferences that B^t_tF has made. A B^t_tF inference can be identified because the text contains 'B^t_tF says... '. If a user double clicks an element from the list, the 'More information' window (see Figure 4.8) is shown with the corresponding information for the selected declaration. This section of the user interface has two buttons:
  - The 'Sort' button orders alphabetically the facts by the name of the declaration.
  - The 'Delete Fact' button allows a user to retract a fact; it is not available for inferences. Let us say an assignment log contains four facts and one inference: fact 1, fact 2, inference 1, fact 3 and fact 4. If fact 2 is deleted, inference 1, fact 3 and fact 4 will be removed from the list, then B^t_tF's next declaration to assign would be that of fact 2. After a user has entered the corrected assignment, B^t_tF will make the corresponding inferences, inference 1 might reappear or not, and then B^t_tF will verify if fact 3 is still valid, if it is, it will be re-entered automatically. The same for fact 4.
- **Bubble 6** points to 'Upload State' button, which allows a user to upload a BttF.csv file.[5]

[4] See Inference 3.
[5] Only a BttF.csv is read at this point.

- **Bubble 7** points to `'Save State'` button, which allows a user to download a `BttF.csv` file with current feature assignments, and a feature model file (`.bttf`). The files generated are saved in the following path:

  < `user_home` > `\Desktop\BttF\` < `program_name` > `\` < `yyyyMMdd hh.mm` > `\`

  This path format allows a user to store several states of the partitioning process.

- **Bubble 9** points to the `'Annotate code'` button, which is enabled once all declarations have been assigned to a feature. This button executes the modification of the program code base, feature annotations are added to the code. More details about this are explained in next section.

The final list of facts and inferences for Operations' program is contained in Table 4.2. A user provided 19 facts, BttF inferred the assignment of 9 declarations, 47% of the assignments were inferred by BttF.

BttF also identified two hooks: `opl.OPL.main` and `opl.OPLTest.test`. In the inferences made with fact 4, can be found that BttF identified these hooks and also decided to remove its current assignment, this according to caveats of Inference 1. In fact 9, a user said that container `opl.OPLTest` was `fprivate` of `BASE`, assigning hook `opl.OPLTest.test` to `BASE` by inference, this in accordance with caveats of Inference 2. Later, BttF asked about hook `opl.OPL.main`'s assignment in fact 15.

4.3 BttF's Output

For both BttF's execution modes, a `.csv` file with feature assignments can be downloaded by a user from BttF's user interface. This file contains every declaration contained in a program's CRG and its current feature assignment.

For recursive partitioning, a declarations assignment file will be saved according to the line of the feature model that is in process. For example, consider again feature model in Equation (4.1). If a user completed assignments according to the first and second line of this feature model, the assignments file will only present features from this set: {`S`, `T`, `B`, `C`}.

When BttF's is executed in its Eclipse plugin mode, BttF is capable to modify a program's source code accordingly to a feature assignments. When a program is partitioned for SPL purposes, BttF presents an option to annotate the program's source code.

4.3.1 R4 for Java Code Feature Annotation

Java is not feature-aware. But it does have declaration annotations. Kim et al developed R4, a new plug-in for Eclipse that supports annotation-based Java SPLs and the OO refactoring of such SPLs [50].

#	Fact/Inference
1	opl is fpublic of BASE
2	opl.Exp is fpublic of BASE
3	opl.Exp.eval() is fprivate of EVAL
	opl.OPL.main(String[] args), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
	opl.OPLTest.test(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
	opl.Plus.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
	opl.Times.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
	opl.Int.eval(), BttF says since it calls fprivate opl.Exp.eval() THEN it also belongs to EVAL
4	opl.Exp.print() is fprivate of PRINT
	opl.Plus.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
	opl.Times.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
	opl.Int.print(), BttF says since it calls fprivate opl.Exp.print() THEN it also belongs to PRINT
	opl.OPL.main(String[] args), BttF says it's hook because calls element(s) classified in future features
	Hook opl.OPL.main(String[] args), BttF says it calls fprivate elements, its feature needs to be determined
	opl.OPLTest.test(), BttF says it's hook because calls element(s) classified in future features
	Hook opl.OPLTest.test(), BttF says it calls fprivate elements, its feature needs to be determined
5	opl.Int is fpublic of BASE
6	opl.Int.v is fpublic of BASE
7	opl.Int.Int(int a) is fprivate of BASE
8	opl.OPL is fpublic of BASE
9	opl.OPLTest is fprivate of BASE
	Hook opl.OPLTest.test(), BttF says it's a child of opl.OPLTest THEN it also belongs to BASE
10	opl.Plus is fpublic of BASE
11	opl.Plus.l is fpublic of BASE
12	opl.Plus.r is fpublic of BASE
13	opl.Plus.Plus(Exp L, Exp R) is fprivate of BASE
14	opl.Times is fpublic of BASE
15	opl.OPL.main(String[] args) is fpublic and hook of BASE
16	opl.Times.l is fpublic of BASE
17	opl.Times.r is fpublic of BASE
18	opl.Times.Times(Exp L, Exp R) is fprivate of BASE
19	test is fprivate of BASE
	test.Util, BttF says it's a child of test THEN it also is fprivate of BASE
	test.Util.val(int r), BttF says it's a child of test.Util THEN it also is fprivate of BASE

Table 4.2: Operations Program Partitioning. Facts and Inferences.

Here is how R4 works: A *configuration* is a set of features that uniquely specifies a product of an SPL. R4 uses a Java annotation type called @Feature. Within it defines a boolean constant for every feature. The value is true if the feature is selected, false otherwise. Figure 4.10 is a @Feature declaration with three features $\{A, B, C\}$ that encodes configuration $\{A, B, \neg C\}$. The legality of a configuration is specified by a feature model [1]. R4 supports the GUIDSL or FeatureIDE specifications of feature models [4, 96].

Every declaration of an R4 codebase has a @Feature annotation with a feature expression as an argument. Feature-guarded code blocks are surrounded by if(feature-expr) statements. If the expression is true for the configuration, that declaration or code block is included in the target program, otherwise it is removed.

Figure 4.11a is a @Feature-annotated codebase where class foo and field t belong to feature A. Feature B adds method inc. Feature C adds fields r, s and guards an update to field

```java
@interface Feature {
 static final boolean A = true;
 static final boolean B = true;
 static final boolean C = false;

 boolean value();
}
```

Figure 4.10: The `@Feature` Annotation Type

`r`. Figure 4.11b is the projected codebase for configuration {A, B, ¬C}. A reader can observe how the code segments for features `A` and `B` are present, but not the code segments for feature `C`. `R4` uses code folding to accomplish projection. For more details, see [50].

```java
@Feature(A)
class foo {
 @Feature(A)
 int t;

 @Feature(C)
 int r=0, s=7;

 @Feature(B)
 void inc(){
  if(C){
   r=r+2;
  }
  t=t+3;
 }
}
```

(a) SPL Code base

```java
@Feature(A)
class foo {
 @Feature(A)
    int t;




    @Feature(B)
    void inc(){



     t=t+3;
    }
}
```

(b) Code base for Config A,B,¬C

Figure 4.11: `@Feature` Annotations and an `R4` Projection

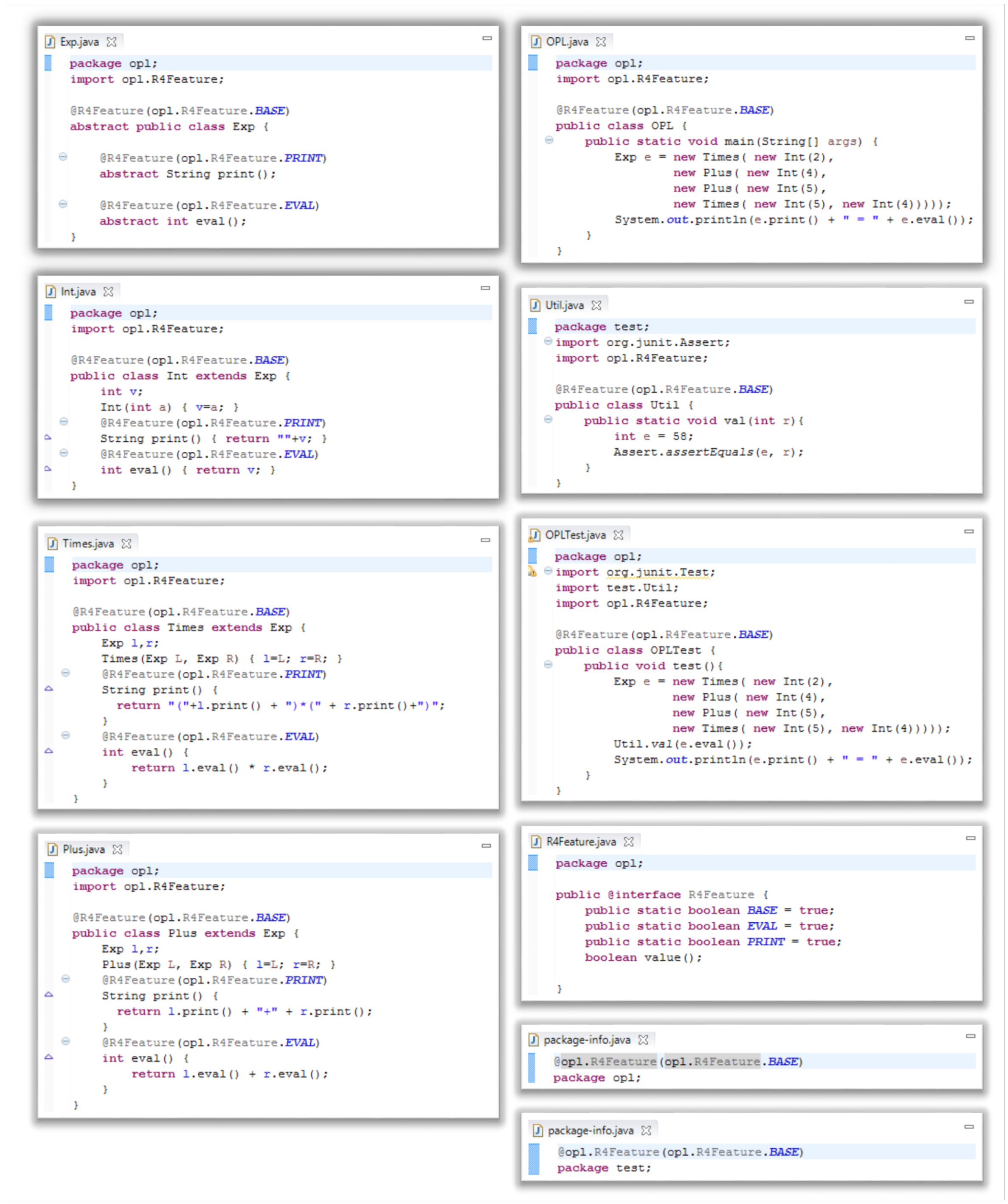
4.3.2 BttF's R4 Application

BttF assigns a feature to every declaration of a program[6]. We use `R4` to `@Feature`-annotate each declaration. What we cannot do is automatically introduce `if(feature-expr)` statements to feature-modularize code within method bodies, such as the body of `inc()` in Figure 4.11a. Feature-refactoring method bodies is a difficult problem that requires deep knowledge of program semantics and must be done manually. For example, in case of hooks, we identify them, but a programmer must manually feature-refactor them.

In Figure 4.12 is possible to observe our example program `Operations`, which was partitioned in the previous section. `Operations` source code now contains `R4Feature` anno-

[6] Identified in a program's CRG.

tations according to the assignments made through `BttF`. A new Java `@interface` type, named `R4Feature`, has also been added to the project. This `R4` SPL configuration file has all existing features selected (`BASE`, `EVAL` and `PRINT`). In Figure 4.13 is also possible to observe that this `@interface` has been added to the program. `Operations` code base has been converted into an annotated SPL.



Exp.java
```java
package opl;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
abstract public class Exp {

    @R4Feature(opl.R4Feature.PRINT)
    abstract String print();

    @R4Feature(opl.R4Feature.EVAL)
    abstract int eval();
}
```

OPL.java
```java
package opl;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class OPL {
    public static void main(String[] args) {
        Exp e = new Times( new Int(2),
                new Plus( new Int(4),
                new Plus( new Int(5),
                new Times( new Int(5), new Int(4)))));
        System.out.println(e.print() + " = " + e.eval());
    }
}
```

Int.java
```java
package opl;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class Int extends Exp {
    int v;
    Int(int a) { v=a; }
    @R4Feature(opl.R4Feature.PRINT)
    String print() { return ""+v; }
    @R4Feature(opl.R4Feature.EVAL)
    int eval() { return v; }
}
```

Util.java
```java
package test;
import org.junit.Assert;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class Util {
    public static void val(int r){
        int e = 58;
        Assert.assertEquals(e, r);
    }
}
```

Times.java
```java
package opl;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class Times extends Exp {
    Exp l,r;
    Times(Exp L, Exp R) { l=L; r=R; }
    @R4Feature(opl.R4Feature.PRINT)
    String print() {
      return "("+l.print() + ")*(" + r.print()+")";
    }
    @R4Feature(opl.R4Feature.EVAL)
    int eval() {
        return l.eval() * r.eval();
    }
}
```

OPLTest.java
```java
package opl;
import org.junit.Test;
import test.Util;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class OPLTest {
    public void test(){
        Exp e = new Times( new Int(2),
                new Plus( new Int(4),
                new Plus( new Int(5),
                new Times( new Int(5), new Int(4)))));
        Util.val(e.eval());
        System.out.println(e.print() + " = " + e.eval());
    }
}
```

Plus.java
```java
package opl;
import opl.R4Feature;

@R4Feature(opl.R4Feature.BASE)
public class Plus extends Exp {
    Exp l,r;
    Plus(Exp L, Exp R) { l=L; r=R; }
    @R4Feature(opl.R4Feature.PRINT)
    String print() {
      return l.print() + "+" + r.print();
    }
    @R4Feature(opl.R4Feature.EVAL)
    int eval() {
        return l.eval() + r.eval();
    }
}
```

R4Feature.java
```java
package opl;

public @interface R4Feature {
    public static boolean BASE = true;
    public static boolean EVAL = true;
    public static boolean PRINT = true;
    boolean value();

}
```

package-info.java
```java
@opl.R4Feature(opl.R4Feature.BASE)
package opl;
```

package-info.java
```java
@opl.R4Feature(opl.R4Feature.BASE)
package test;
```

Figure 4.12: `Operations` Java program source code annotated.

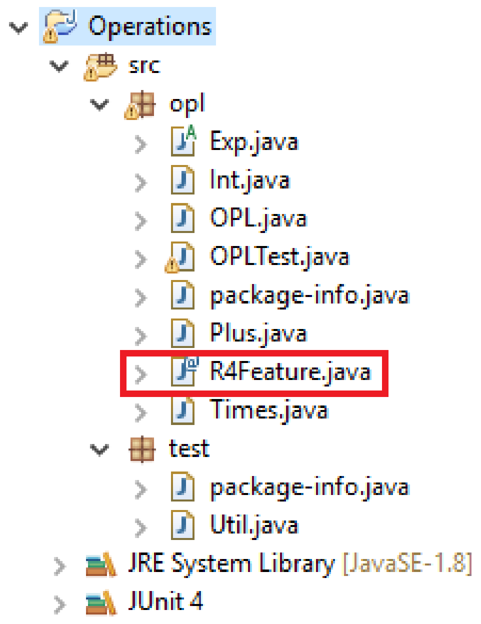


Figure 4.13: `Operations` Java program structure with added `R4Feature`.

Chapter 5

B$^{t}_{t}$F Partitioning of Java Programs into SPLs

5.1 Experimental Setup

Partitioning a Java program into an SPL is the main scenario for which B$^{t}_{t}$F was designed. To evaluate B$^{t}_{t}$F, we could have used programs that have never been converted into an SPL. However, we needed existing SPLs as ground truth to validate B$^{t}_{t}$F – to reproduce existing feature definitions. Even this was not easy as we had to convert their codebases into plain Java programs. In short, our reasons for selecting these programs were:

- Using programs that were Java SPLs as ground truth, and
- Ensures that our experimental results are reproducible.

We selected seven Java programs to evaluate B$^{t}_{t}$F. Three factors that were influential in determining which programs to use for this evaluation were:

- They needed to be *known Java SPLs*. In other words, they were commonly used in SPLs experiments so that others could more easily reproduce our results.
- Since we needed to convert them from SPLs into plain Java Programs, we picked *annotated SPLs* for which we had access to the source code and its feature assignments.
- When partitioning a program, B$^{t}_{t}$F might find that some provided facts are invalid. *Expert knowledge* is required to provide a correction of invalid facts. We chose programs that either we were well-versed in their design or that we could become experts.

Table 5.1 lists the programs selected for B$^{t}_{t}$F evaluation.[1] Programs are sorted by `LOC` (Lines

[1] The source code for these programs is available in https://github.com/priangulo/BttFTestProjects

of Code)[2]. `# of Features` is the number of features contained in a program's partitioning feature model. `# of Declarations` refers to the number of declarations found in a program's CRG. `# of Packages` and `# of Classes` refer to the count of these type of declarations.

Program	LOC	# of Features	# of Declarations	# of Packages	# of Classes
mixin	35489	16	3953	1	501
unmixin	33895	11	3703	1	500
guidsl	16307	25	1818	1	147
bali2jak	12773	9	1220	1	135
Prevayler	7915	7	1210	22	157
Gates	1235	4	146	5	20
Operations	80	3	28	2	7

Table 5.1: Java programs to validate B^t_tF partitioning into SPLs.

5.2 Research questions

B^t_tF is an inferencing engine. Our first two research questions are related to its inferencing performance. The last question is related to the behavior of the number of possibilities shown to a user during the partitioning process. Let the inference rate be:

$$\textit{inference rate} = \frac{\texttt{\# of B}^{t}_{t}\texttt{F inferences}}{\texttt{\# of program declarations}} \tag{5.1}$$

RQ1 In which scenario(s) is B^t_tF's inference rate the highest?

RQ2 In which scenario(s) is B^t_tF's inference rate the lowest?

RQ3 Is there a correlation between the assignment time and the number of possibilities? In other words, does the number of possibilities shrink as more facts are introduced?

The following section presents the results obtained with the performed experiments and the answers to our research questions.

5.3 Results

For every program used in this evaluation, we obtained a feature model and a full list of assignments. These assignments were validated by a program expert. B^t_tF's task was to ingest facts, report errors, and infer the rest.

Table 5.2 contains general information about the partitioning results obtained with each program. We performed four experiments on each program:

[2] We used `JavaLOC` tool[24] to obtain the number of lines of code.

1. **E1 (Original)**. For the first experiment we used the feature model provided with each program.
2. **E2 (Top Heavy)**. For the second experiment we used a feature model with only two features: BASE and TOP, where all of the original features, except for the first one, were comprised in TOP.
3. **E3 (Base Heavy)**. For the third experiment we also used a feature model with only two features: BASE and TOP, where all of the original features, except for the last one, were comprised in BASE.
4. **E4 (Balanced)**. For the fourth experiment we used the same two-features feature model. Now, half of the original features were comprised in BASE, and the other half in TOP.[3] [4]

The fields contained in Table 5.2 are as follows:

- Experiment is the experiment number.
- N is the number of features in the partitioning feature model.
- FP is the number of fprivate facts provided by an expert.
- FP% is the percentage of FP,

$$\mathtt{FP\%} = \frac{\texttt{\# of fprivate facts}}{\texttt{\# of program declarations}} \cdot 100 \tag{5.2}$$

- wFP% is the weighted percentage of fprivate facts provided by an expert. Not all fprivate facts produce the same number of inferences. The number of inferences produced by a fprivate fact depends on the number of declarations that reference that fprivate declaration. For this reason, it is necessary to have a weighted percentage of fprivate facts. The *weight* is the number of declarations that reference a declaration d, this concept was previously introduced as RefTo(d). We use RefTo(d) as a weight because the declarations contained in it would be assigned to d's feature, if d is fprivate.

$$\mathtt{wFP\%} = \frac{\sum \texttt{\# of fprivate facts weight}}{\sum \texttt{\# of all declarations weight}} \cdot 100 \tag{5.3}$$

- I is the overall number of declarations that were inferred. For this scenario, B$^{\mathtt{t}}_{\mathtt{t}}$F makes five types of inferences. These are mentioned in later bullet points.
- I% is the percentage of declarations inferred by B$^{\mathtt{t}}_{\mathtt{t}}$F,

$$\mathtt{I\%} = \frac{\texttt{\# of B}^{\mathtt{t}}_{\mathtt{t}}\texttt{F inferences}}{\texttt{\# of program declarations}} \cdot 100 \tag{5.4}$$

[3] For feature models with an uneven number of features, the largest half was TOP.

[4] For Operations program there is no Experiment 4, as Operations original feature model only had three features, so Experiment 4 would have been the same than Experiment 2 or 3.

- `IF` is the number of declarations inferred because they reference a `fprivate` method or field.
- `IC` is the number of declarations inferred because they belong to a `fprivate` container.
- `IL` is the number of declarations that were assigned `fprivate` because they belong to the last feature in the feature model.
- `IO` is the number of declarations inferred because they had only one feature in bounds.
- `IN` is the number of declarations inferred because they have no references other than to its container.
- `H` is the number of identified `hook` methods.
- `ANP` is the average of number of possibilities seen during the assignment process.
- `SNP` is the standard deviation of `ANP`.

5.3.1 RQ1. In which scenario(s) BttF's inference rate is the highest?

From our experiments, we learned that BttF's reached its highest inference rates when more declarations were assigned to the last layer. This corresponds to E2 (Top Heavy). The average inference rate obtained with E2 experiments is 71.82. The results obtained in E2 experiments is highlighted in light gray in Table 5.2. Table 5.3 contains the number of declarations assigned to the last layer, and the inference rate obtained per experiment:

- `LS%` is the percentage of declarations assigned to the last feature,

$$\text{LS\%} = \frac{\texttt{\# of program declarations assigned to last feature}}{\texttt{\# of program declarations}} \cdot 100 \qquad (5.5)$$

- `I%` is BttF's inference rate.

E2 (Top Heavy) was the experiment in which we comprised the declarations originally assigned to all features except the first one. As can be observed for E2, the highest the number of declarations assigned to the last feature, the highest the inference rate obtained.

To statistically support the previous premise, we used the *Pearson Correlation Coefficient* (ρ). ρ is a measure of dependency between two variables [58]. ρ can take any value from -1 to $+1$. 1 represents a total positive linear correlation (as one of the variables increases the other too increases), 0 is no linear correlation, and -1 represents a total negative linear correlation (as one of the variables increases the other decreases). In this case we are measuring the correlation between `LS%` and `I%`. We performed this calculation using R-Project,

Experiment	N	FP	FP%	wFP%	I%	I	IF	IC	IL	IO	IN	H	ANP	SNP
mixin														
E1	16	8	0.20%	10.02%	30.9%	1222	419	785	18	0	0	9	18.86	11.36
E2	2	0	0.00%	0.00%	82.0%	3540	5	0	3235	0	0	5	1.48	0.75
E3	2	8	0.20%	10.02%	30.9%	1222	419	785	18	0	0	0	2.73	0.45
E4	2	8	0.20%	10.02%	32.1%	1269	424	785	60	0	0	5	2.70	0.46
unmixin														
E1	11	113	3.05%	8.43%	28.6%	1061	753	299	9	0	0	5	12.93	7.01
E2	2	0	0.00%	0.00%	84.5%	3128	6	0	3422	0	0	4	1.42	0.71
E3	2	113	3.05%	8.43%	28.6%	1061	753	299	9	0	0	0	2.75	0.43
E4	2	113	3.05%	8.43%	29.5%	1093	759	299	35	0	0	1	2.74	0.44
guidsl														
E1	25	7	0.38%	4.26%	16.9%	304	107	174	23	0	0	44	28.21	16.31
E2	2	0	0.00%	0.00%	80.6%	1449	0	0	1449	0	0	0	1.74	0.89
E3	2	7	0.38%	4.26%	16.9%	304	107	174	23	0	0	1	2.84	0.39
E4	2	7	0.38%	4.26%	45.5%	818	183	174	461	0	0	46	2.35	0.79
bali2jak														
E1	9	19	1.55%	7.52%	26.1%	319	155	146	18	0	0	27	10.27	5.56
E2	2	0	0.00%	0.00%	67.9%	828	1	0	827	0	0	1	1.67	0.87
E3	2	19	1.55%	7.52%	26.1%	319	155	146	18	0	0	1	2.79	0.40
E4	2	19	1.55%	7.52%	41.8%	510	155	146	209	0	0	6	2.45	0.76
Prevayler														
E1	7	0	0.00%	0.00%	37.9%	458	0	0	458	0	0	12	7.87	5.06
E2	2	0	0.00%	0.00%	64.9%	785	21	0	764	0	0	11	1.87	0.90
E3	2	0	0.00%	0.00%	37.9%	458	0	0	458	0	0	1	2.34	0.90
E4	2	0	0.00%	0.00%	45.8%	554	21	0	533	0	0	10	2.17	0.90
Gates														
E1	4	3	1.96%	0.47%	18.5%	28	1	2	25	0	0	6	5.52	1.88
E2	2	2	1.32%	0.47%	37.1%	56	7	2	47	0	0	6	2.38	0.64
E3	2	3	1.96%	0.47%	18.5%	28	1	2	25	0	0	0	2.60	0.62
E4	2	3	1.96%	0.47%	22.5%	34	1	2	31	0	0	0	2.54	0.62
Operations														
E1	3	6	21.42%	9.41%	35.7%	10	5	2	3	0	0	2	4.21	1.13
E2	2	5	17.85%	3.52%	35.7%	10	1	2	7	0	0	1	2.82	0.47
E3	2	6	21.42%	9.41%	35.7%	10	5	2	3	0	0	2	2.67	0.61

Table 5.2: Java SPLs partitioning results.

which is an environment for statistical computing [81]. The function used to obtain ρ is `cor` from the statistical package `corrplot` [102]. The result obtained is:

$$\rho(\text{LS\%}, \texttt{I\%}) = \mathbf{0.93} \tag{5.6}$$

Since `E2` experiments present the lowest numbers of `fprivate` facts[5] (most of them are zero), then we know that `fprivate` facts are not a strong factor to determine the inference

[5] Please see columns `FP` and `FP%` in Table 5.2

E1		E2		E3		E4	
LS%	**I%**	**LS%**	**I%**	**LS%**	**I%**	**LS%**	**I%**
mixin							
0.5%	30.9%	89.5%	82.0%	0.5%	30.9%	3.3%	32.1%
unmixin							
0.3%	28.6%	92.0%	84.5%	0.3%	28.6%	1.9%	29.5%
guidsl							
2.1%	16.9%	85.5%	80.6%	2.1%	16.9%	29.4%	41.8%
bali2jak							
2.5%	26.1%	76.1%	67.9%	2.5%	26.1%	19.4%	41.8%
Prevayler							
39.5%	37.9%	67.1%	64.9%	39.5%	37.9%	46.4%	45.8%
Gates							
19.2%	18.5%	41.7%	37.1%	19.2%	18.5%	25.8%	22.5%
Operations							
14.3%	35.7%	32.1%	35.7%	14.3%	35.7%	-	-

Table 5.3: RQ1. Last feature size vs. BᵗₜF's Inference Rate.

rate obtained with them. This is supported by calculating ρ using only E2 experiments numbers, the result obtained is:

$$\rho(\texttt{LS\%},\ \texttt{I\%}) = \mathbf{0.99} \tag{5.7}$$

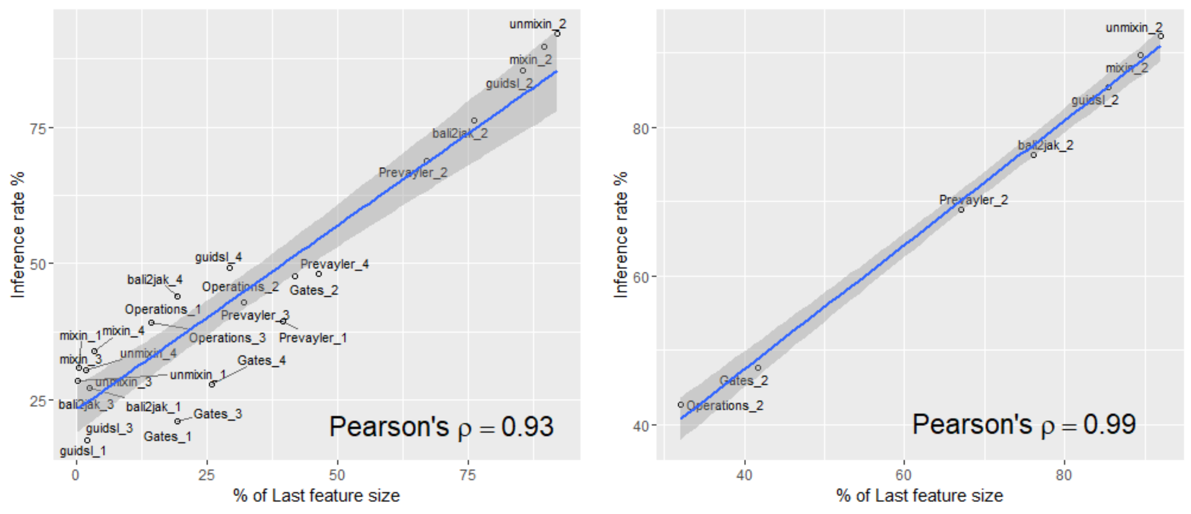
These results indicate a *high positive linear correlation* between our compared variables (LS%, I%). This statistically supports that a higher percentage of declarations assigned to the last feature, leads to a higher BᵗₜF's inference rate.

A visual representation of these results can be observed in Figure 5.1. Figure 5.1a shows the results when all the experiments are considered. Figure 5.1b shows the results when only E2 experiments are considered. These charts contain a point per experiment. The blue trend line marks a total positive correlation ($\rho = 1$), with a 95% confidence region highlighted in gray.

In Figure 5.1a it is possible to observe that the experiments corresponding to E1, E3 and E4 are more apart from the trend line than E2 experiments. The reason for this is because their inference rates are not completely due to the size of the last layer (LS%). The amount of fprivate facts were a factor that influenced the inference rate.

In Figure 5.1b is possible to observe that Gates and Operations are at the bottom of the trend line. For E2 experiments we comprised in TOP feature, all the original features but the first one, which was assigned as BASE. However, Gates and Operations had most of their declarations in their first feature. Gates had 58.3% of its declarations in BASE. Operations had 67.9% of its declarations in BASE. This caused their TOP features to be relatively small, therefore their inference rate was also low.

The main lesson learned from these results is that to better take advantage of BᵗₜF's inferencing capabilities, it is recommendable to partition a program in a recursive manner, two features at a time.



(a) All experiments. (b) Only E2 experiments

Figure 5.1: RQ1. Correlation between Last feature size percentage and B$^{t}_{t}$F's Inference rate.

5.3.2 RQ2. In which scenario(s) B$^{t}_{t}$F's inference rate is the lowest?

In Table 5.2 we can observe that `E1` and `E3` experiments obtained the lowest inference rates in comparison with other experiments. We can also observe that for every program, the inference rates obtained with `E1` and `E3` are the same for every program because of two factors:

- `E1` last feature and `E3` last feature are the same. They comprised the same declarations. For `E1` experiments we used the original feature model, for `E3` we comprised all the features but the last one in `BASE`, and the last feature was assigned as `TOP`. Therefore the amount of inferences obtained thanks to assigning declarations to the last layer (see `IL` column in Table 5.2) was the same for both `E1` and `E3`.

- Same effect of `fprivate` facts. For `E1` all `fprivate` facts were provided for all features except for the last one, for which is not necessary to provide `fprivate` facts, as all the declarations in the last layer are automatically assigned as `fprivate` by B$^{t}_{t}$F. For `E3` experiments, the `BASE` feature comprised all the features in the original feature model, but the last one, then it also contained all the provided `fprivate` facts.

Therefore, the factors that determine the inference rate were the same for `E1` and `E3`, causing that their results were the same.

The reason why `E1` and `E3` inference rates were the lowest is because they mostly depended on `fprivate` facts, which were few and with the capacity of impacting a few number of other declarations, this capacity being determined by their *weight*[6]. This can be observed in Table 5.2, column `wFP%`, which shows `fprivate` facts allowed `B`$^{t}_{t}$`F` to infer a feature for:

- 10% of `mixin` declarations,
- 8% of `unmixin` declarations,
- 4% of `guidsl` declarations,
- 7% of `bali2jak` declarations,
- 0% of `Prevayler` declarations (because no `fprivate` facts were provided),
- $\frac{1}{2}$% of `Gates` declarations, and
- 9% of `Operations` declarations.

Does this mean that `fprivate` facts are not useful? No. In Table 5.2 in `FP` and `FP%` columns, we can observe the small proportion of `fprivate` facts provided, in comparison with the total number of declarations. And yet, for `E1` and `E3` experiments this `fprivate` facts are the main reason for the inference rates obtained.

The main lesson learned from these results is that it would be convenient to provide to a user the list of declarations sorted by their weight. This way a user might be able to provide more impactful `fprivate` facts. However, this does not change the fact that `fprivate` facts need to be provided carefully. For example, if a user decided to assign a Java `package p` as `fprivate` of feature `F`, all the declarations contained in `p` would recursively be assigned to `F`, which might not be a user's desired intention.

5.3.3 RQ3. Is there a negative correlation between Assignment Time and Number of Possibilities?

We are looking for a negative correlation between Assignment Time and Number of Possibilities. *Assignment Time* refers to the order in which a declaration of a program `p` was assigned. The first declaration that is assigned has an assignment time of 1. The last declaration to be assigned has an assignment time of the length of declarations in `p`. *Number of possibilities* refers to the number of options[7] that were in bounds when a declaration was assigned.

To say that we are looking for a *negative correlation* between Assignment Time and Number of Possibilities means that we expect that as Assignment Time increases, the Number of Possibilities decreases.

[6] As mentioned before, weight is the number of declarations that reference a declaration `d` (`RefTo(d)`), and thus they would be assigned to `d`'s feature, if `d` is `fprivate`.

[7] For more details please see Section 3.3.1.

The Pearson Correlation Coefficient (ρ) is also used for this research question. In this case we calculate the correlation between declarations' assignment time (`AT`) and the number of possibilities (`NP`) each of the declarations had at the moment of assignment.

We obtained a ρ per each experiment performed. To obtain an overall result, we cannot simply average the obtained coefficients, this would lead to an spurious result, because these coefficients depend on the data variances, and since they were obtained from different experiments, they have different variances. In order to average the results, it is necessary to standardize the resulting correlation coefficients variances. To do this standardization, we apply *Fisher's Z-transformation* to ρ. Fisher's Z-transformation is a method that transforms a ρ value to a Z (normal) distribution, therefore allowing to perform aggregation operations over these coefficients (i.e. getting a mean). The obtained values can then be averaged. The resulting average then is transformed again into ρ form to provide an accurate *average correlation coefficient* for all the experiments [60, 103].

We again performed these calculations using R-Project, which is an environment for statistical computing [81]. The function used to obtain ρ is `cor` from the statistical package `corrplot` [102]. The function used to transform ρ to a Z distribution is `fisherz`, and the function to transform it back to its Pearson correlation form is `fisherz2r`, both functions belong to the statistical package `psych` [84]. Table 5.4 contains the ρ coefficients obtained with every experiment (ρ(`AT,NP`)) and their corresponding fisher-transformed values (`fisherZ`(ρ)).

E1		**E2**		**E3**		**E4**	
ρ(`AT,NP`)	`fisherZ`(ρ)	ρ(`AT,NP`)	`fisherZ`(ρ)	ρ(`AT,NP`)	`fisherZ`(ρ)	ρ(`AT,NP`)	`fisherZ`(ρ)
mixin							
0.32	0.34	0.20	0.20	0.31	0.32	0.26	0.26
unmixin							
0.06	0.06	0.13	0.13	0.03	0.03	0.02	0.02
guidsl							
-0.02	-0.02	0.20	0.20	0.24	0.25	0.39	0.42
bali2jak							
0.02	0.02	0.58	0.66	0.13	0.13	0.39	0.41
Prevayler							
0.50	0.55	0.46	0.50	0.66	0.80	0.61	0.71
Gates							
0.24	0.25	0.21	0.22	0.35	0.36	0.26	0.26
Operations							
-0.11	-0.11	-0.15	-0.15	-0.33	-0.35	-	-

Table 5.4: RQ2. Correlation between Assignment Time and Number of Possibilities.

Those fisher-transformed values were averaged, the result is: `0.239`[8]. As mentioned before, this average is now converted back into `rho`, the obtained value is: `0.235`, which is the overall correlation factor observed between Assignment Time and Number of Possibilities seen across experiments.

[8] We used all the digits of the resulting numbers.

0.235 indicates a low positive linear correlation between our compared variables (AT, NP). We were looking for a negative linear correlation. The interpretation is that a late assignment time of a declaration does not guarantee a fewer number of available possibilities. In other words, *the number of possibilities is not steadily shrinking as more facts are introduced*.

The rationale behind this, is that a declaration's assignment possibilities are affected *only* by the assignments of the declarations it references, and not by the overall assignments made. B$^{t}_{t}$F's possibilities calculation is in certain way greedy, as it traverses a CRG and calculates possibilities for one declaration at a time.

Does this mean that feature possibilities are never reduced? No. In Figure 5.2 it can be observed, for all the experiments, the number of possibilities each declaration had. For example, the first chart in Figure 5.2a corresponds to mixin's Experiment 1. The very first line in that chart represents the first declaration that was assigned, which had 31 possibilities[9] The higher the bar the more possibilities, the lower the bar the fewer possibilities. It is observable that there was a reduction of possibilities for some declarations, but this reduction was not constant across assignment time, it rather fluctuated.

Tables 5.5 to 5.8 present data that shows the percentage of declarations that had a reduction on their possibilities. In these tables we have bins to group the number of possibilities that declarations had (Possibilities bin), the average percentage of declarations that were on that bin (AVG), and the standard deviation (SD) of that average.

For E1 experiments (shown in Table 5.5) we have four bins:

- Bin 1 contains declarations that had at most one third of all available possibilities. For example, if for a program the maximum number of possibilities is 9, this bin would contain those declarations that had at most 3 possibilities.
- Bin 2 contains declarations that had at most two thirds of all available possibilities.
- Bin 3 contains declarations that had over two thirds of all available possibilities, but less than the maximum number of possibilities.
- Bin 4 contains declarations that had all possibilities. This group had no reduction in their possibilities.

E2-E4 experiments (shown in Tables 5.6 to 5.8) had the same feature model (P : BASE TOP), thus, the maximum number of possibilities was 3[10]. For these experiments we have three bins:

- Bin 1 contains declarations that had only one possibility.

[9] mixin's feature model used in Experiement 1 has 16 features, this multiplied by 2 (fprivate and fpublic), minus 1 because the last feature only has fprivate as a possibility, gives 31 as the maximum number of possibilities that could be available for a declaration.

[10] The possibilities are: BASE fprivate, BASE fpublic and TOP fprivate

- `Bin` 2 contains declarations that had two possibilities.
- `Bin` 3 contains declarations that had three possibilities. This group had no reduction in their possibilities.

Possibilities bin	**AVG**	**SD**
1	23.1%	8.5%
2	28.3%	6.8%
3	27.8%	9.7%
4	28.7%	19.8%

Table 5.5: `E1`. Average of declarations % per Number of Possibilities' Bin.

Possibilities bin	**AVG**	**SD**
1	44.9%	27.6%
2	18.6%	11.6%
3	36.5%	24.5%

Table 5.6: `E2`. Average of declarations % per Number of Possibilities' Bin.

Possibilities bin	**AVG**	**SD**
1	7.6%	11.0%
2	19.0%	6.8%
3	74.5%	7.2%

Table 5.7: `E3`. Average of declarations % per Number of Possibilities' Bin.

Possibilities bin	**AVG**	**SD**
1	13.1%	13.0%
2	24.1%	5.9%
3	62.8%	8.8%

Table 5.8: `E4`. Average of declarations % per Number of Possibilities' Bin.

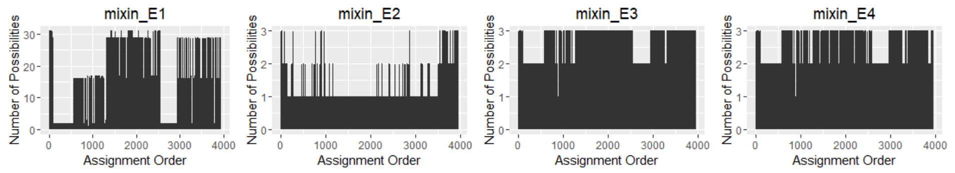
For `E1` Experiments (Original), on average, over 70% of the declarations had a reduction in their possibilities. And over 20% of the declarations had at most a third of all possibilities available.

For `E2` Experiments (Top Heavy), on average, over 60% of the declarations had a reduction in their possibilities. And over 44% of the declarations had one possibility available.

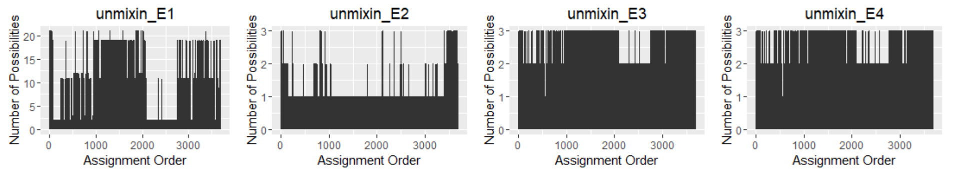
For `E3` Experiments (Base Heavy), on average, around 25% of the declarations had a reduction in their possibilities. Only 7.6% of the declarations had one possibility available.

For `E4` Experiments (Balanced), on average, around 27% of the declarations had a reduction in their possibilities. Only 13.1% of the declarations had a single possibility available.

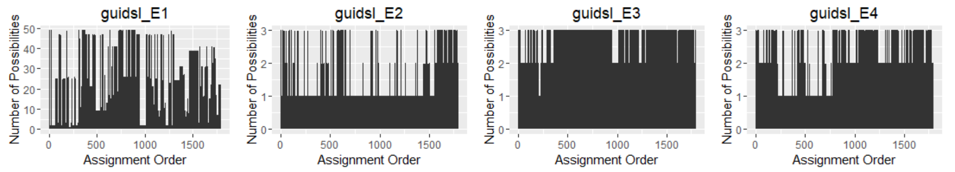
Even though the number of possibilities are not linearly reducing as more facts are introduced, `BttF` does provide a reduction of the possibilities for at least 25% of the declarations, this in the worst case, and up to 70% in the best case.



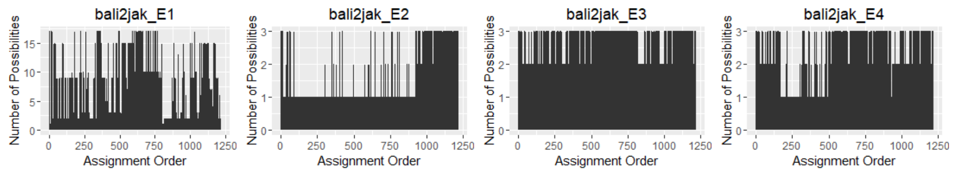
(a) Assignment Order vs. Number of Possibilities in `mixin` Experiments



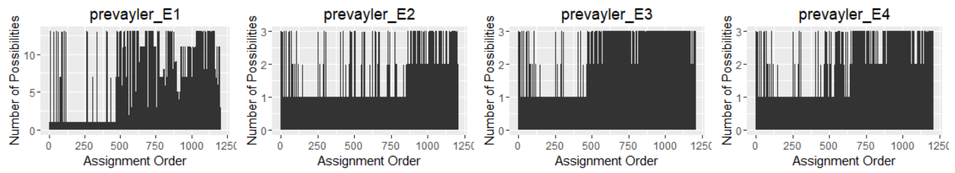
(b) Assignment Order vs. Number of Possibilities in `unmixin` Experiments



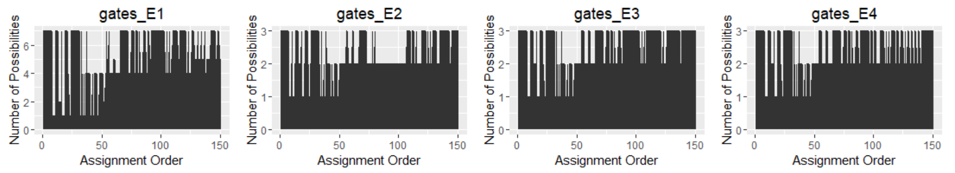
(c) Assignment Order vs. Number of Possibilities in `guidsl` Experiments



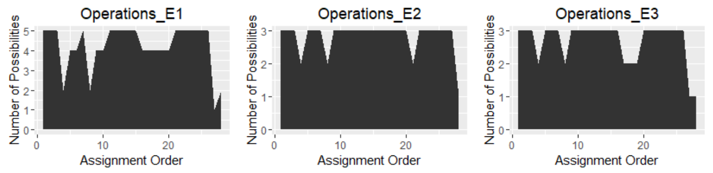
(d) Assignment Order vs. Number of Possibilities in `bali2jak` Experiments



(e) Assignment Order vs. Number of Possibilities in `Prevayler` Experiments



(f) Assignment Order vs. Number of Possibilities in `Gates` Experiments



(g) Assignment Order vs. Number of Possibilities in `Operations` Experiments

Figure 5.2: Assignment Order vs. Number of Possibilities

5.4 Related Work

In this section we present three works that are related to `B`$^{t}_{t}$`F`. We could not compare numerically these works' results with PT's results, for two reasons:

- They partially extract features from a program, instead of fully partitioning a program into features. Therefore, they have as a concern to measure how much of a feature's code was successfully located, whereas for `B`$^{t}_{t}$`F`, every declaration is assigned to a feature, therefore, all code is successfully located.
- They support the process of assigning code to features, however they do not have formal automatic inferencing, as `B`$^{t}_{t}$`F` has. Therefore they do not present inference rates, or an equivalent concept, that we can compare numerically to. Inference rate is the main metric that we use to measure `B`$^{t}_{t}$`F`'s performance.

These related works use recall and precision as their success measures. In general, *recall* refers to the ability of locating relevant material given a topic; *precision* refers to how much of the relevant material obtained was correctly classified [97]. What is defined as *relevant* or as *correctly classified* depends on the evaluation approach taken in each of these related works. Never-the-less, using these definitions of recall and precision, we can assert that `B`$^{t}_{t}$`F` has a 100% recall, and 100% precision. `B`$^{t}_{t}$`F` has 100% recall because it finds all declarations of each feature (not just some of one feature). `B`$^{t}_{t}$`F` also has 100% precision because it guarantees that each declaration is assigned to a correct feature. The following subsections present these related works in chronological order.

5.4.1 Valente-2012

Valente proposes and describes a semi-automatic approach to annotate the code of optional features [99]. This approach was developed to annotate Java programs. An *optional feature* is defined as "a feature that can be safely removed without disrupting the behavior of the core".

As with `B`$^{t}_{t}$`F`, this approach requires of an expert to provide a set of initial facts, which they call *seeds*. These seeds behave as `B`$^{t}_{t}$`F`'s `fprivate` facts.

Their approach is composed of two phases:

1. **Propagation**. Declarations that correspond to the seeds are annotated with the feature provided with the seed. Also, all the declarations that reference the seeds are annotated with the feature of the seed. However, they have defined a set of rules to limit the propagation of the seeds according to their declaration type. Their rules apply to these Java declaration types: `package`, `class`, `interface`, `method`, `field`, `local variable` and `formal parameter`. A similar behavior occurs with `B`$^{t}_{t}$`F`'s `fprivate` facts, but `B`$^{t}_{t}$`F` does not have such specific rules to limit propagation.

2. **Expansion**. Their algorithm checks whether a feature annotation can be expanded to its enclosing lexical context. *Lexical context* means the surrounding code of a code fragment that has been annotated. This algorithm consists of a loop where semiautomatic expansions are calculated and proposed to a user, a user can either accept these proposals or reject them. The rules used to determine semiautomatic expansions are as follows:

   - **Rule E1**. The feature of a loop or conditional statement can be expanded to the statement's body.
   - **Rule E2**. The feature used in the body of a `loop` or conditional statement can be expanded to the statement's expression, if this expression does not produce side effects. An expression has a *side effect* when it updates the program state.
   - **Rule E3**. The feature used in an `else` statement, must be expanded to include its clause. Also, the feature used in the expression of a `return` must be expanded to include its full statement.
   - **Rule E4**. The feature used in the body of a method must be expanded to its signature.
   - **Rule E5**. The feature used by all members of a class must be expanded to the class declaration, and to any use of the class.
   - **Rule E6**. The feature used in the left-hand side of an assignment must be expanded to include its right-hand side.

   The role of this expansion phase, is similar to some inferences made by B^t_tF, but applied at a more granular level. B^t_tF does not support the assignment of parts of a method's body, or a field's assignment statement.

To evaluate their approach they used three SPLs. As we did with B^t_tF, they used programs that were SPLs to have a ground truth:

- Prevayler, they use the monolithic, non-SPL-based version of the program, which has 2974 lines of code[11]. They extracted 5 optional features[12].
- JFreeChart, with 91174 lines of code. They extracted 3 optional features.
- ArgoUML, with 117983 lines of code. They extracted 4 optional features.

They measure recall and precision in their experiments. *Recall* measures whether their approach was able to identify all the code of a feature. *Precision* measures whether their approach was able to annotate all the relevant code of a feature.

They presented their results per extracted feature of each program, reproduced in Table 5.9.

[11] The version of `Prevayler` they used is different from the version we used to evaluate B^t_tF.
[12] Those are the same features we considered when evaluating B^t_tF, plus `BASE` and `TOP`

Feature	**Precision**	**Recall**
Prevayler		
`Monitor`	100%	100%
`Censorship`	100%	100%
`Replication`	100%	94%
`Snapshot`	100%	59%
`GZip`	100%	100%
JFreeChart		
`Pie Charts`	100%	99%
`3D Charts`	100%	99%
`Statistical Charts`	100%	99%
ArgoUML		
`State Diagram`	87%	88%
`Activity Diagram`	94%	96%
`Design Critics`	99%	96%
`Logging`	89%	97%

Table 5.9: Valente's results.

Their approach reached an average precision of 97%, and an average recall of 94%. With BᵗₜF we have a precision of 100% and a recall of 100%, plus the inferences made by BᵗₜF, which do not require user interaction.

5.4.2 ICFL-2013

Peng et al propose an "Iterative Context-aware approach to automatic Feature Location (ICFL)" [76]. This work is related to ours on the topic of mapping a set of declarations to features of a feature model. ICFL measures how likely is the mapping between a feature and a set of declarations based on their structural and lexical similarity. This work relies on *Information Retrieval (IR)*, which is a field of study that given a search query, finds data that is related or relevant in a large collection of unstructured data sources, for example, text files [20].

Structural similarity measures how much of a feature model structure can be mapped to a program's declarations. By *feature model structure* they refer to the dependencies, containments, and interactions[13] among features in a feature model. Their goal is to map a feature `F`, which has interactions and dependencies with other features, to a part of a program, that resembles these same interactions and dependencies with other parts of the program. In turn, these other parts of the program would have to satisfy the structures of the features they are being mapped to.

Lexical similarity measures the similarity between a feature description and the source

[13] Feature `F` *depends* on feature `G`, if it requires of `G`. Feature `F` *contains* feature `H` if `H` is a subfeature of `F`. There is an *interaction* between feature `F` and feature `M`, if one of them modifies the other's behavior, or if they are alternative features (one or the other).

code of a declaration, this similarity is computed using IR techniques. A *feature description* is an attribute that ICFL captures, which contains a description about a feature.

ICFL is fed with already established feature-element mapping pairs, these are equivalent to the facts that B^t_tF obtains from an expert, then it fills in the remaining assignments. They evaluated their approach by performing experiments on two programs:

- Linux kernel is an open source operating system kernel written in C. It has 12 million lines of code and 700 features.
- DirectBank is a proprietary program provided by their industry partner. It has 30 thousand lines of code, and 71 features.

They measure recall and precision on their experiments. *Recall* measures the ability to identify declarations that should belong to a feature of a feature model, for all features. *Precision* measures if a declaration was assigned to the right feature.

Their results show a high recall and low precision. For DirectBank, the best precision obtained was 58% with a recall of 91%. This means they identified 91% of the declarations that should be mapped to the features of a feature model, and from those declarations, they mapped 58% of them to a correct feature. For Linux kernel the best precision obtained was 29% with a recall of 66%.

They start with the premise that not all of a program's declarations necessarily belong to a feature.[14] We, on the contrary, partition a program and assign every declaration to a feature. Therefore, B^t_tF's recall is 100%.

For B^t_tF's evaluation, all declarations were assigned to a correct feature, therefore our precision is always 100%. What is there to measure for B^t_tF is its inference rate. The number of facts that ICFL receives, or how they impact the rest of the declarations assignment, is not stated, therefore we cannot numerically compare their results with ours.

5.4.3 LEADT-2014

Kästner, Dreiling, and Ostermann present LEADT (Location, Expansion, And Documentation Tool) [48]. LEADT is a semiautomatic tool for extracting features from a legacy Java program. LEADT works by extracting a *feature at a time*.

The use of LEADT requires an expert to provide a feature model and initial *seeds*, which are equivalent to the facts provided to B^t_tF. However, LEADT receives a single seed per feature. LEADT supports the location of code fragments of a feature, by recommending probable code fragments from the legacy Java program. They describe it as guiding a user into looking in the right location to find *all* fragments of a feature.

LEADT is therefore a recommendation system. It recommends code fragments that are likely to belong to the searched feature. It relies on three recommendations mechanisms:

[14] In effect, their BASE is not considered a feature.

- **Type System**. This is their key recommendation mechanism. It looks up references within the source code. A *reference* represents a relationship between two code fragments, it has a source and a target, for example a method invocation is the source, whereas this same method's declaration is the target. If the target `t` of a reference is annotated with a feature `F`, but the source `s` of that same reference has not been annotated with `F`, their Type System would issue a prioritized recommendation to annotate `s` with `F`.

- **Topology Analysis**. As we did with B^t_tF, they also produce a graph to represent a program's declarations and their relationships. As B^t_tF's CRG, LEADT's graph also is created using *Abstract Syntax Trees* (ASTs) provided by Eclipse. They adapted Robillard's topology analysis [85] to follow their graph representation of a program and produce a ranked list of code fragments that are likely to belong to the searched feature. To produce these ranks they use the metrics specificity and reinforcement. *Specificity* ranks higher those declarations that refer to (or are referred from) only a single declaration, and ranks lower those declarations that refer to (or are referred from) many elements. *Reinforcement* ranks higher those declarations that refer to (or are referred from) many already annotated declarations, this based on the idea that they are probably part of a cluster of feature code.

- **Text Comparison**. They implemented their own text comparison mechanism to determine the similarity of a feature's vocabulary and a declaration's vocabulary. A *declaration's vocabulary* is obtained by tokenizing all the words contained in a declaration's name. A *feature's vocabulary* is composed of all of its declarations' vocabularies. For example, if many declarations of feature `F` contain in their names the word *lock*, then their text comparison recommendation system would recommend other code fragments that contain that word.

For each declaration, LEADT obtains an overall recommendation priority `w` by merging the recommendations obtained with its three recommendation mechanisms. LEADT produces a list of declarations ordered by `w`. This list shows, in order of recommendation, the declarations that are more likely to belong to the searched feature. Every time a user accepts a recommendation to add a declaration to the searched feature, LEADT recalculates its recommendation list.

To evaluate LEADT, four SPLs were used. As we did with B^t_tF, they used programs that were SPLs to have a ground truth:

- Prevayler, with 8009 lines of code and 5 features[15].

- MobileM, with 4653 lines of code and 7 features.

- Lampiro, with 44584 lines of code and 10 features, although for LEADT evaluation they searched only for 2 features.

[15] We also used `Prevayler` to evaluate B^t_tF. The version we found of `Prevayler` had slightly less lines of code. The feature model that we used to partition `Prevayler` had 7 features, the original 5 features, plus `BASE` and `TOP`.

- Sudoku, with 1975 lines of code, and 5 features.

They measure recall and precision in their experiments. *Recall* measures which percentage of all the code of a feature was located. For example, if a feature had 10 declarations, and they found 9, their recall would be 90%. *Precision* measures which percentage of the recommendations was correct. LEADT overall results are an average recall of 97%, this means 97% of all code per feature was located; and an average precision of 42%, which means that 42% of all the recommendations given were correct. B^t_tF fully partitions a program into features, which means our recall is 100%. And as mentioned before, our precision is also 100%.

LEADT shares with B^t_tF the use of Eclipse's ASTs to produce a graph representation of a program in question, and also the use of facts provided by an expert.

LEADT supports a user's decision to add or not a declaration to a feature. B^t_tF not only guides a user in assigning every declaration to a feature, it reduces a user's work by inferencing assignments.

Chapter 6

BttF Refactoring a Program into a Framework and Plug-in

6.1 Introduction

We use the term *framework* to denote an *Object Oriented* (OO) application framework, which is a reusable and semi-complete application that can be specialized to produce custom applications. Frameworks are targeted to specific application domains [37].

Frameworks are a popular and effective software reuse technique. Some examples of well-known or currently popular application frameworks are: .NET[1], Spring[2] and Java AWT[3]. Creating a framework requires extensive understanding of the application domain where it would be applied, and intensive design experience writing programs in this domain.

Frameworks can be classified according to the technique used to extend them. They range from purely *white-box* to *black-box*. A *white-box* framework relies heavily on OO inheritance and other OO features. To extend a white-box framework, classes from the framework need to be inherited and extended, and methods that serve as hooks in the framework, have to be overridden. A *black-box* framework supports extensibility by defining interfaces that allow to plug components into the framework using object composition [37].

Designing and implementing a framework is a non-trivial task. Most of the challenges in creating an application framework are related to determine how generic a framework should be. The more generic a framework, the more work is required to implement an extension. The more concrete a framework, the less work is required to implement an extension [10]. And perhaps it would require more maintenance to keep adapting it to possible

[1] .NET is a framework for developing applications that leverage Microsoft technologies and services, `https://www.microsoft.com/net/`.

[2] Spring is a framework for enterprise application development based on Java, `http://projects.spring.io/spring-framework/`.

[3] Java AWT is a framework for creating user interfaces for Java programs, `https://docs.oracle.com/javase/9/docs/api/java/awt/package-summary.html`.

changes in its domain [15].

We used `BttF` as a support tool to partition Java programs into a white-box framework and a plug-in. We use the term *plug-in* to refer to a framework extension. At a first glance, partitioning a program into a framework and plug-in is the same than partitioning it into two features: `FRAMEWORK` and `PLUGIN`. However, to have a program transformed into a framework and a plug-in requires structural changes: *it requires the separation of framework code from plug-in code*. Examples of structural changes are: move a `class` from a `package` to another, modify a declaration's modifiers, and so on.

These structural changes impose new constraints that are not present when partitioning a program into a SPL. The transformation of a program into a SPL can be made via annotations, which are not as invasive as the changes required to transform a program into a framework and plug-in. Given this, `BttF` needed to consider the constraints imposed by a structural transformation of a program into a framework and plug-in.

This is the first work that automates almost all tasks in the partitioning of Java program into framework and plug-in. The following sections present how this was achieved, the constraints that we needed to take into consideration, and the obtained results.

6.2 Java Constraints for Framework and Plug-in Refactoring

We discovered four constraints when partitioning and transforming Java programs into a framework and plug-in. They were added to `BttF` as rules and are explained next.

C1 Superclasses cannot be partially in the framework and in the plug-in.
A *superclass*, or non-terminal class, is a class that is extended by at least another class in the program[4]. Please observe the class diagram in Figure 6.1a. It contains two classes `A` and `B`. `A` has two members, field `d` and method `m`. `B` has three members, field `f`, and methods `m` and `p`. `B` extends `A`, therefore `A` is a superclass. Let say a user makes the following assignments:

- `A.d` is assigned to `FRAMEWORK`.
- `A.m` is assigned to `PLUGIN`.
- `B.f` is assigned to `FRAMEWORK`.
- `B.m` is assigned to `FRAMEWORK`.
- `B.p` is assigned to `PLUGIN`.

Given these assignments, both classes `A` and `B` are partially in the framework and in the plug-in. A class in this situation is transformed into a framework and plug-in

[4] This also applies to Java `interfaces`.

structure by following these general steps[5], we use class `A` to explain these steps:

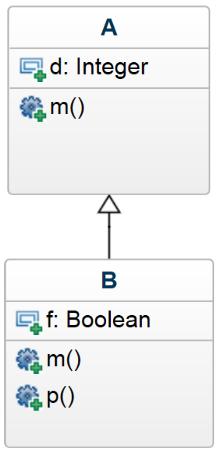
(a) An abstract class `A` is created in the framework package, call it `fw.A`.

(b) The original class `A` is moved to the plug-in package, call it `pl.A`.

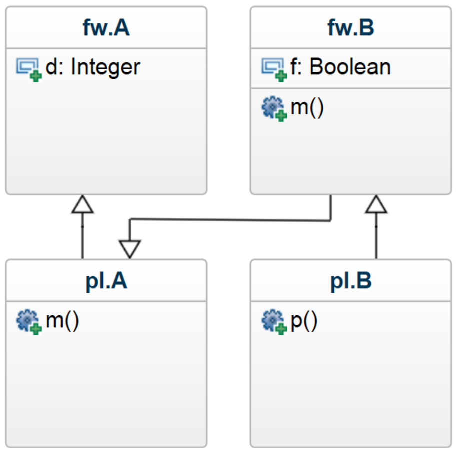
(c) Class `pl.A` extends class `fw.A`.

(d) The members of `A` that were assigned to the framework are pulled-up into `fw.A`, in this case this member is field `d`, and all other members remain in `pl.A`.

When the previous steps are applied to `A` and `B`, we get the result of Figure 6.1b. Since `A` is a superclass that got partially assigned to the framework and the plug-in, it caused an undesired dependency: `fw.B` extends `pl.A` — that is, a framework class extends a plug-in class. The framework cannot have such a dependency in an extension. Therefore, to prevent this situation, a superclass has to either fully belong to the framework or to the plug-in, it cannot be partially in both. We implemented this as a rule in B$^{t}_{t}$F when partitioning a program into framework and plug-in.



(a) Before partitioning.



(b) After partitioning.

Figure 6.1: C1. Problem with partitioning superclasses.

C2 Annotations and Enums cannot be partially in the framework and in the plug-in. A Java `annotation` is a form of metadata. It provides data about a declaration or a code segment [70]. A Java `enum` is a data type that represents a fixed set of constants [69]. As mentioned in **C1**, when a class `C` is partially in the framework and in the plug-in, it has to be divided into two classes: an abstract counterpart in the framework `fw.C`, and another class in the plug-in `pl.C`, which extends `fw.C`. Java `annotations` and `enums` do not support `extends`, therefore, they have to either fully belong to the framework or to the plug-in, they cannot be partially in both. We implemented this as a rule in B$^{t}_{t}$F when partitioning a program into framework and plug-in[6].

[5] Section 6.5.2 provides full details about the rules and process for transforming a program into framework and plug-in.

[6] For SPLs, features are denoted via Java `annotations`, therefore this constraint does not apply for SPL partitioning of `enums` and `annotations`.

C3 Static methods cannot be hooks.
Java `static` is a modifier. When it is applied to a method, the method can be invoked without the need of creating an instance of its declaring class [71].

If `BttF` identifies method `m` from class `C` as `hook`[7], then it would be transformed into a framework and plug-in structure by following these steps:

(a) An abstract method `m` is created in the framework counterpart of `C`, this method is now `fw.C.m`. This new method has the same signature than original `m` plus an `abstract` modifier.

(b) Method `m` in plug-in counterpart of `C`, `pl.C.m`, overrides `fw.C.m`.

(c) The original implementation of `m` is now contained in `pl.C.m`[8].

It is not possible in Java to have a method with both `static` and `abstract` modifiers, therefore, `static` methods cannot be hooks and cannot be transformed accordingly. To handle this situation, if a `static` method `m` is assigned to the `FRAMEWORK` feature, `BttF` infers that all the declarations that `m` references should also belong to the `FRAMEWORK`.

C4 Methods of non-terminal classes in the `FRAMEWORK` cannot be hooks.
A `hook` method forces a class to be partially in the framework and in the plug-in. If method `m` of class `C` is a hook, it has to have an abstract signature in `C`'s framework counterpart `fw.C`, while `C`'s plug-in counterpart `pl.C` contains `m` original implementation.

Because of constraint **C1**, a superclass or non-terminal class in the framework cannot be partially in the framework and in the plug-in, therefore, if `superclass C` is assigned to the `FRAMEWORK` feature, and one of its members method `m` is identified by `BttF` as a `hook`, then `BttF` infers that all the declarations that `m` references should also belong to the `FRAMEWORK`.

6.3 Experimental Setup

To evaluate `BttF` as a support tool to partition a Java legacy program into a framework and a plug-in, we selected programs that were known in framework and design pattern research community.

We selected six Java programs to evaluate `BttF`. Four were taken from Schmidt's LiveLessons course [91][9]. The other two were taken from Batory's undergrad course on Software Design

[7] A `hook` is a method that references declarations introduced in a later feature. In this scenario we have only two features `FRAMEWORK` and `PLUGIN`, therefore all `hook`s would be in `FRAMEWORK`.

[8] A manual task after `BttF` is done is for a programmer to decide what fraction of each concrete hook method is to be moved into the framework.

[9] The four programs we used are used in his undergrad course of Patterns and Frameworks.

[7]: `Gates`[10], and `Calc`. We used `Gates` earlier to evaluate `BttF` on partitioning a program into a SPL. `Calc` is a small calculator program that was designed to be partitioned into framework and plug-in.

Table 6.1 lists the programs selected for `BttF` evaluation[11]. Programs are sorted by `LOC` (Lines of Code)[12]. `# of Declarations` refers to the number of declarations found in a program's CRG. `# of Packages` and `# of Classes` refer to the count of these type of declarations.

Program	**LOC**	**# of Declarations**	**# of Packages**	**# of Classes**
ExpressionTree	4793	555	1	84
ImageStreamGangApp	2630	211	3	36
ImageTaskGangApplication	2457	232	1	33
Gates	1235	141	5	19
SearchTaskGang	1058	91	1	12
Calc	476	66	3	5

Table 6.1: Java programs to validate `BttF` partitioning into Framework and Plug-in.

6.4 Research Questions

For the scenario of partitioning a program into a framework and plug-in, we formulated two research questions. The first is related to `BttF`'s inference rate and the second evaluates the possibility of automatically transforming a program into framework and plug-in using the assignments obtained with `BttF`.

RQ1 Is `BttF`'s inference rate better for framework and plug-in partitioning, than for the best scenario in SPL partitioning?

This question compares the inference rate obtained with framework and plug-in partitioning, versus the best scenario obtained with SPL partitioning. The best scenario would be experiments `E2`, for which all programs were partitioned using a two features feature model: `BASE` and `TOP`, with a top heavy configuration[13].

RQ2 Is it possible to automatically transform a Java program into a framework and plug-in structure?

With *framework and plug-in structure*, we refer to the separation of a framework's code fragments from a plug-in's code fragments.

[10] We used a version of Gates that had no annotations.

[11] The source code for these programs is available in https://github.com/priangulo/BttFTestProjects

[12] We used `JavaLOC` tool [24] to obtain the number of lines of code.

[13] Top heavy means most of the declarations were assigned to the `TOP` feature.

6.5 Results

To partition the programs listed in Section 6.2 into framework and plug-in, we became familiar with their code, classes hierarchies and declarations dependencies. We aimed for a *top heavy framework*, we tried to assign most of the declarations to `FRAMEWORK` feature[14].

For the programs we took from LiveLessons [91], we defined which part of the source code should go to the framework and which to the plug-in, based on:

- The class diagrams, which served as a footprint to *draw a line* among the classes and identify which functionality could be core and therefore part of the framework.
- Comments and naming conventions were useful to identify framework and extension code segments.

In the case of `Gates`, it originally has four features: `BASE`, `TABLE`, `CONSTRAINT` and `EVAL`. We comprised `BASE` and `TABLE` in the framework, and the rest on plug-in. For `Calc`, which is a small program, we used its class hierarchy to define what should belong to framework, and what to plug-in.

Table 6.2 contains general information about the partitioning results obtained with each program. The fields of this table are:

- `Program` is the name of the program.
- `FN` is the number of declarations assigned to `FRAMEWORK` feature.
- `PN` is the number of declarations assigned to `PLUGIN` feature.
- `FP` is the number of `fprivate` facts provided by an expert.
- `FP%` is the percentage of `FP`,

$$\texttt{FP\%} = \frac{\texttt{\# of fprivate facts}}{\texttt{\# of program declarations}} \cdot 100 \qquad (6.1)$$

- `I` is the overall number of declarations that were inferred. The types of inferences B$^{t}_{t}$F makes in this scenario are mentioned in later bullet points.
- `I%` is the percentage of declarations inferred by B$^{t}_{t}$F,

$$\texttt{I\%} = \frac{\texttt{\# of B}^{t}_{t}\texttt{F inferences}}{\texttt{\# of program declarations}} \cdot 100 \qquad (6.2)$$

- `IF` is the number of declarations inferred because they reference a `fprivate` method or field.

[14] `FRAMEWORK` feature is equivalent to `BASE` feature in SPL partitioning.

- `IC` is the number of declarations inferred because they belong to a `fprivate` container.
- `IL` is the number of declarations that were assigned `fprivate` because they belong to the last feature in the feature model.
- `IO` is the number of declarations inferred because they had only one feature in bounds.
- `IN` is the number of declarations inferred because they have no references other than to its container.
- `NT` is the number of declarations that belong to a non-terminal `class` and `B`^t_t`F` overrode their assignments, because their declaring class cannot be partitioned so some of their members are in the framework, and the rest in the plug-in[15, 16].
- `AE` Is the number of declarations that belong to a Java `annotation`, or a Java `enum`. These types cannot have some members in the framework and the rest on the plug-in[17, 18].
- `SM` is the number of declarations that are referenced by a `static` method of a framework class. Since this method cannot be a `hook` all the declarations it references must belong to the framework too[19].
- `NM` is the number of declarations that are referenced by a method of a non-terminal framework class. Since this method cannot be a `hook` all the declarations it references must belong to the framework too[20].
- `H` is the number of identified `hook` methods.

Program	**FN**	**PN**	**FP**	**FP%**	**I%**	**I**	**IF**	**IC**	**IL**	**IO**	**IN**	**NT**	**AE**	**SM**	**NM**	**H**
`ExpressionTree`	372	183	1	0.2%	32.9%	183	0	0	154	9	0	0	0	13	7	0
`ImageStreamGangApp`	152	59	0	0.0%	34.1%	72	0	0	45	1	0	0	6	17	3	0
`ImageTaskGangApplication`	103	129	0	0.0%	48.7%	113	0	0	107	1	0	0	0	0	5	0
`Gates`	107	34	0	0.0%	44.0%	62	0	0	28	1	0	0	0	29	4	1
`SearchTaskGang`	79	12	14	15.4%	44.0%	40	6	21	10	0	0	0	0	0	3	0
`Calc`	32	34	0	0.0%	33.3%	22	7	0	15	0	0	0	0	0	0	9

Table 6.2: Java Framework and Plug-in partitioning results.

[15] For more details please see Section 6.2.

[16] If a user's original assignment did not go against this rule, `B`^t_t`F` does not override it, and therefore it does not count as `NT`.

[17] For more details about this rule please see Section 6.2.

[18] If a user's original assignment did not go against this rule, `B`^t_t`F` does not override it, and therefore it does not count as `AE`.

[19] For more details about this rule please see Section 6.2.

[20] For more details about this rule please see Section 6.2.

6.5.1 RQ1. Is B$^{t}_{t}$F' s inference rate better for framework and plug-in partitioning than for the best scenario in SPL partitioning?

The highest inference rate obtained when evaluating B$^{t}_{t}$F on SPL partitioning, was with E2 experiments[21]. These experiments were performed with a feature model containing two features: BASE and TOP. Most of the declarations were assigned to the TOP feature, we refer to E2 experiments as *top heavy*.

This research question compares the inference rate obtained with framework and plug-in partitioning (FWPL-results), versus the inference rate obtained with E2 experiments (SPL-E2-results). For framework and plug-in partitioning we aimed for a *top heavy framework* configuration, most of the declarations assigned to FRAMEWORK feature[22]. This would be equivalent to a *base heavy* configuration, which is the opposite from what we had in E2 experiments. However, for framework and plug-in partitioning we had additional constraints that could increase the number of inferences, therefore, this research question.

To compare FWPL-results with SPL-E2-results, we need to perform a hypothesis test. A simple comparison of results' means is not enough, as the results come from different experiments. *Hypothesis testing* is a "decision-making process for evaluating claims about a population", this claims are usually related to statistical means and proportions about a population. A hypothesis test uses data obtained from a sample to make a decision about whether a claim can be accepted or rejected [14, p. 400–430]. In this case we are claiming that FWPL-results are better than SPL-E2-results.

The formula to use on hypothesis testing depends on the attributes of the data that we have as a result from our experiments. We have two samples[23], one that represents FWPL-results, and another that represents SPL-E2-results.

Our samples are considered small, for FWPL-results we have six values, and for SPL-E2-results, seven. Given this, we need to use *t-test* as our statistical test [27, p.357, 358]. We compare the means of both samples. And our claim is that FWPL-results mean is greater than SPL-E2-results mean. The formal enunciation is:

$$\begin{aligned} \mathtt{H_0} : \mu &= 64.67, && \text{results obtained with SPL E2 experiments} \\ \mathtt{H_a} : \mu &> 64.67, && \text{claim based on results obtained with FWPL} \end{aligned}$$

To perform this hypothesis test we used R-Project, which is an environment for statistical computing [81]. The function we used is `t.test` from R's core package `stats` [82]. We provided this function with the values for $\mathtt{H_0}$ (SPL-E2-results) and $\mathtt{H_a}$ (FWPL-results), and specified that we were conducting a *greater* comparison, which evaluates if $\mathtt{H_a}$'s mean μ is

[21] For more details please see Section 5.3.

[22] In Table 6.2 note that for all the programs we used in this evaluation, the number of declarations assigned to FRAMEWORK (FN field) is greater than the number of declarations assigned to PLUGIN (PN field); except for `ImageTaskGangApplication`, which the source code contained several plug-ins, which we comprised in PLUGIN feature, therefore, in this case PLUGIN was larger than FRAMEWORK.

[23] We say samples, because we did not conduct exhaustive experimentation, which would imply testing with every existing Java program, which is impossible.

greater than $\mathtt{H}_0$'s mean μ. By default, this test is performed with a confidence value (α) of 0.05%. This means that we have 95% certainty over the decision we make about rejecting (or not) our $\mathtt{H}_0$ hypothesis.

The result obtained was:

$$p\text{-}value = \mathbf{0.991} \tag{6.3}$$

The *p-value* (probability value) is the probability of obtaining $\mathtt{H}_a$ results when $\mathtt{H}_0$ is true. In this case, it is the probability of obtaining the mean stated in $\mathtt{H}_a$ when the true population mean is the one stated in $\mathtt{H}_0$. P-value measures the probability of $\mathtt{H}_a$ results being sufficiently evident and consistently better to disprove the results obtained with $\mathtt{H}_0$. To interpret a p-value we follow these rules [14, p. 418]:

$$\begin{aligned} p\text{-}value < \alpha, &\qquad \text{reject } \mathtt{H}_0 \\ p\text{-}value \geq \alpha, &\qquad \text{do not reject } \mathtt{H}_0 \end{aligned}$$

Since our p-value 0.991 is greater than our confidence value α 0.05, we cannot reject $\mathtt{H}_0$. This means that **we do not have enough evidence to state that the results obtained with FWPL-results are better than the results obtained with SPL-`E2`-results**.

Are they equivalently good? This means, are our FWPL-results as good as the one obtained with SPL-`E2`-results? This question can be answered by performing another hypothesis test, with the following theorems:

$$\begin{aligned} &\mathtt{H}_0 : \mu = 64.67, &\qquad &\text{this is the claim, we want to assert this remains true} \\ &\mathtt{H}_a : \mu \neq 64.67 \end{aligned}$$

We again perform this hypothesis test using R-Project. We use the same function `t.test` from `stats` package [82]. We provided the same values for $\mathtt{H}_0$ (SPL-`E2`-results) and $\mathtt{H}_a$ (FWPL-results), but now, we specified that we were conducting a *two-sided* comparison, which evaluates if $\mathtt{H}_a$'s mean μ is *different* than $\mathtt{H}_0$'s mean μ. By default, this test is performed with a confidence value (α) of 0.05%. This means that we have 95% certainty over the decision we make about rejecting (or not) our $\mathtt{H}_0$ hypothesis.

The result obtained was:

$$p\text{-}value = \mathbf{0.017} \tag{6.4}$$

Since our p-value 0.017 is less than our confidence value α 0.05, we have evidence to reject $\mathtt{H}_0$. This means that **we do not have enough evidence to state that the results obtained with FWPL-results are as good as the results obtained with SPL-`E2`-results**.

Despite the constraints imposed on framework and plug-in partitioning, the inference rate obtained is not better than the inference rate obtained with `E2` experiments on SPL partitioning. This shows clear evidence that the size of the last feature (`TOP` or `PLUGIN`) is the main contributor to `B`^t_t`F`'s inference rate, and the constraints imposed on framework and plug-in partitioning did not produce a considerable improvement over a *top heavy frame-*

work[24] configuration, to make it as good as the results obtained with a *heavy top* configuration, as the one had with SPL E2 experiments.

These hypothesis tests could be performed again with a larger sample, more programs. A larger set of experiments would allow to have a more accurate perspective. However, these results provide an indication that the size of the last feature (TOP or PLUGIN) most directly affects B^t_tF's inference rate.

6.5.2 RQ2. Is it possible to automatically transform a Java program into a framework and plug-in structure?

With *framework and plug-in structure*, we refer to the separation of a framework's code fragments from a plug-in's code fragments. In this case we are separating them into different Java packages. The purpose of this research question was to find and implement an automated process to separate a program into a framework and a plug-in, ensuring that the framework has no dependencies towards the plug-in source code, and that it compiles.

It is possible to use tools to modify the source code of a program. For Java programs, Eclipse offers automated refactorings. A *refactoring* is "the process of changing the structure of software while preserving its external behavior" [62]. These refactoring operations range from simple ones, as renaming a declaration, to more complex ones, like extracting an interface from a class[25]. However, to achieve our desired results, we needed to perform a large number of refactoring operations, systematically, for which Eclipse provides no tool that could achieve this.

The R3 Java refactoring engine, developed by Kim et al, addresses this need of invoking refactorings programmatically [51]. We collaborated with R3 creators to develop a script and the required set of operations that would allow to transform a Java program into a framework and plug-in structure. With B^t_tF we obtained a road map of the structural changes that needed be performed on a program, and through R3 we executed those transformations. A description of the resulting implementation is explained next, among with the results obtained from it.

Transforming a Java program into framework and plug-in using R3

R3 can execute refactoring scripts, where a script is a Java method. A developer can specify a set of refactoring operations to be applied to a program's source code, and R3 performs them without affecting the integrity of the source code. This is achieved by R3 through the creation of an internal database that reflects a Java program's contents [51]:

- Program declarations (i.e. classes, methods, fields), their

[24] Equivalent to *heavy base* in SPL partitioning.

[25] For a full list of available Eclipse refactoring operations see [32].

- Containment relationships, and
- Java language features such as inheritance and modifiers.

In order to use `R3` as our transformation engine and apply the set of assignments discovered with `BttF`, we created an `R3` script that could perform the steps required for this transformation. In this case, `R3` was used as a transformation engine and not a refactoring engine, to see the differences please see Appendix A. The code of this script can be found in Appendix B; *it is an order of magnitude larger than any prior* `R3` *script.* As a reader could observe, the script is essentially a (big) method in the Java language. Our script is a Java program which makes use of `R3`'s `API`. The steps performed on this script are described next:

Step 1. Packages.
We create framework package `fwp`, and rename original package to `plp`. The original packages is now plug-in package.

Step 2. Factories.
We create an empty framework abstract factory class `fwp.factory`, and an empty plug-in concrete factory class `plp.factory`, where `plp.factory` extends `fwp.factory`.

Step 3. Populate framework.
We populate `fwp` with the classes and other types that were assigned to it. We do as follows depending on the type:

3a A concrete class in the framework cannot invoke a constructor of an abstract class in the framework, the Java compiler will throw an error, therefore, we need to make this class abstract. See Step 4 later.

3b A class `C` with only some members that belong to the framework:

(a) Create an abstract class in the framework package, `fwp.C`.

(b) Extend the new abstract class `fwp.C` with the current extends of the concrete class `plp.C`.

(c) Make the concrete class `plp.C` extend the new abstract class `fwp.C`.

3c An interface `I` with only some members that belong to the framework:

(a) Create an interface in the framework package, `fwp.I`.

(b) Extend the new interface `fwp.I` with the current extends of the original interface `plp.I`.

(c) Make the original interface `plp.I` extend the new interface `fwp.I`.

Step 4. Lift constructor calls.
Replace constructor calls with calls to local factories. We apply this to members of types

```
class A {
 void m(Y b){
   new Z(4); //constructor call
   new Y();  //constructor call
 }
}
```

(a)

```
class A {
 void m(Y b){
   new Factory().newZ(4); //factory call
   new Factory().newY();  //factory call
 }
}

class Factory {
 public Z newZ(int x){
   return new Z(x);
 }
 public Y newY(){
   return new Y();
 }
}
```

(b)

Figure 6.2: Lift constructor calls.

that entirely or partially belong to the framework. In every member `m` of a class, replace every constructor call to abstract framework classes, with calls to local factory methods.

Consider method `m` of class `A` in Figure 6.2a. This method has two constructor calls: `Z(int)` and `Y()`. In this case `Z` and `Y` are framework abstract classes, therefore, we create a factory class `Factory`, which contains the factories that would replace the direct constructor calls. `Factory` is shown in Figure 6.2b along with the rewritten method `m`, which no longer calls directly constructors `Z(int)` and `Y()`, but their factories.

We do not apply this to methods identified by BttF as `hooks` because programmers need to refactor them manually. A BttF `hook` is a method `m` that is to undergo a manual *template method* refactoring, where a template (framework-bound) method is factored out of `m` plus a set of hook methods which are to remain in the plug-in and have abstract method counterparts in the framework.

To exemplify why we do not automatically refactor BttF's `hooks`, please observe Figure 6.3, method `m` of class `C` belongs to `FRAMEWORK` and it is a `hook`, it references method `print_value1` from class `X` (line 4) that belongs to `PLUGIN`. These facts can be detected by BttF, however, line 8 also belongs to `PLUGIN`, but since this line does not reference other declaration, there is no way for BttF to automatically detect this. Cases like this show that template method refactoring requires a manual intervention to refactor it correctly.

Step 5. Pull classes and interfaces apart.
We move to the framework the declarations that belong to it:

5a If an entire type belongs to framework, we move it to `fwp`.

5b Pull apart a class or interface that partially belongs to the framework. Do as follows with its members:

```
public class C{
 public void m(int val){
  if(val > 2){
   System.out.println("Ext1: " + X.print_value1(val)); //PLUGIN
  }
  if(val > 5){
   val = val + 2;
   System.out.println("Ext2:" + (val + 3)); //PLUGIN
  }
 }
}

public class X {
 static int print_value1(int val){
  return val + 1;
 }
}
```

Figure 6.3: Problem refactoring a `hook` method.

(a) For a method `m` identified by B^t_tF as `hook`, create an abstract method in the framework. The original code of `m` remains in the plug-in. A developer must decide later how to restructure its contents. Figure 6.4a shows method `m`, which has been identified by B^t_tF as a `hook`. In Figure 6.4b `m` is rewritten. An abstract method `m` is now defined in the abstract framework counterpart of class `A`, `fwp.A`. The original contents of `m` remain in concrete method `plp.A.m`.

(b) For a public constructor, create a factory method in the concrete factory `plp.factory`. Lift the return and parameter types of this new factory method and create a corresponding abstract factory method in the framework abstract factory `fwp.factory`.

Figure 6.5a shows a public constructor `A.A`, class `A` belongs to the framework, therefore, a factory needs to be created for `A.A`. Figure 6.5b shows both framework factory class `fwp.Factory`, and plug-in factory class `plp.Factory`. `fwp.Factory` has an abstract factory method that returns an instance of `A`. `plp.Factory` has a concrete factory method that creates an instance of `A`. Both the concrete and the abstract factories have their return and parameter types lifted: they specifically return `fwp.A`, and expect `fwp.X` as parameter.

(c) For the rest of a class or interface members, that belong to the framework, lift the types they reference and promote them to their corresponding framework abstract class or interface.

Modifying the code that would correspond to an application calling a framework and plug-in, was out of the scope of this partitioning and refactoring process. The code that would correspond to the application part was comprised in the plug-in, and was no subject to application refactoring [10].

We applied this refactoring script to the six programs we used for evaluation. The changes

```java
class A {
 void m(int value){ //m is a hook
  if (value == 2) {
   ...
  }
 }
}
```

(a)

```java
package fwp;
abstract class A {
 abstract void m(int value);
}

package plp;
class A extends fwp.A {
 void m(int value){
  if (value == 2) {
   ...
  }
 }
}
```

(b)

Figure 6.4: Framework+Plugin structure of a `hook` method.

```java
package fwp;
class A {
 public A(X x){
  ...
 }
}
```

(a)

```java
package fwp;
abstract class Factory {
 abstract public fwp.A newA(fwp.X x);
}

package plp;
class Factory extends fwp.Factory {
 public fwp.A newA(fwp.X x){
     ...
    }
}
```

(b)

Figure 6.5: Framework+Plugin structure for a `public` constructor.

these programs underwent can be observed in their class diagrams, which are in Appendix C[26], for every program we have a class diagram that shows its original structure and another class diagram that shows the new structure obtained after applying our `R3` framework and plug-in transformation script.

We successfully applied our transformation script to all six evaluation programs. To verify these programs we ran regression tests:

- `ExpressionTree` program did not have regression tests implemented, we created seven regression tests, to verify each of the available operations.
- `ImageStreamGangApp`, `ImageTaskGangApplication` and `SearchTaskGang` programs had a test class implemented. The tests ran successfully before and after applying the transformations.
- `Gates` and `Calc` had `JUnit`[89] tests, which ran successfully before and after applying the transformations.

The execution time of this refactoring script is almost instantaneous, the maximum time observed was approximately 2 seconds. Performing these refactoring operations one a time until arriving to the desired structure is extraordinarily time consuming and error prone for even the smallest programs [51]. For a program of the size of `ExpressionTree`, it could easily take several days of person-work. Specifics about the increased speed seen by using refactoring scripts versus manually applying refactoring operations, can be seen in the work of Kim et al [51].

As an answer to this research question, we can assert that it is possible to automatically transform a Java program into a framework and plug-in structure.

6.6 Related Work

As mentioned earlier, this is the first work that nearly fully automates the tasks required to partition and restructure a Java program into framework and plug-in. Therefore, this section contains the cornerstone research done by Opdyke[68] and others, that provided with the means to make possible work like ours. Then we mention other research that aim to locate pieces of code that could be generalized.

6.6.1 Cornerstone Research

Twenty-five years ago (1992), William Opdyke completed the first thesis on OO refactorings [68]. The title of his thesis was "Refactoring Object Oriented Frameworks". His research

[26] The original source code for these programs is available in https://github.com/priangulo/BttFTestProjects. The refactored code resulting of applying our transformation script can be found in https://github.com/priangulo/BttFResults/tree/master/FWPl

compiled a list of refactorings that were used in refactoring (Smalltalk) frameworks in the Choices operating system, largely to pull them apart into more manageable modules.

Opdyke introduced the concept of 'refactorings': automatic, semantics preserving program transformations. His thesis did not include a refactoring tool that would automate the refactorings he identified. Therefore, there was no script, and no analysis, by which declarations of Smalltalk programs would be assigned to a framework or to a plug-in. There was no automatic tool to partition a legacy program into a framework or plug-in.

It may come as a surprise that the first really useful refactoring engines appeared in the early 2000s; Eclipse version 1.0 included these refactoring operations: `extract method`, `rename`, and `move` [39, 106]. Eclipse matured over the next 15 years, but still there was no mainstream refactoring engine (such as Eclipse) that supported refactoring scripts. There had been many attempts previously (see [49] for a detailed history), but the first tool that could be used by common programmers or undergraduate students was `R2` in 2015 [49]. In that paper, it was shown that many design patterns simply could not be fully implemented by Eclipse refactorings, which were never designed to be called programmatically in scripts. The basic problems were:

- lack of separation of concerns (`move method` refactorings were coupled with optimizations that removed unneeded parameters – but creating visitors required no such optimizations be invoked);
- the need for more general refactorings (the Eclipse inline refactoring was found to be too restrictive to be useful in scripts, and Eclipse `move method` refactorings were unable to move methods that invoked `super`, where in fact, move methods-with-`super` is useful);
- Eclipse refactorings are slow: each takes about 1/2 second. When hundreds of refactorings are invoked, wait times are simply too long to be interactive; and
- Eclipse refactorings are bug-infested [49, 51].

The necessary tools to script refactorings/transformations in Java appeared only recently. Program transformation systems were certainly available in this time frame, such as Stratego [107] and DMS [13]. These systems are monuments of engineering prowess, their learning curves are measured in weeks or months (which actually is no different than learning the Eclipse refactoring engine) [49]. Other work on Domain-Specific Languages, which attempt to provide lighter-weight language front-ends to manipulate abstract syntax trees still have overhead and have a limited following – Eclipse plugins, which R3 and X15 are examples, seem to be more popular.

6.6.2 Generalization Research

Given that tools to script refactorings/transformations in Java appeared only recently, the related work we mention in here is directed to the topic of when and where to refactor programs to give them better structure. With no exceptions that we are aware, these papers were not accompanied or built-upon existing refactoring engines, so any claims to demonstrable automation are extraordinarily rare.

A recent observation by someone in the refactoring community supports this viewpoint [8]:

> "The refactoring community focused on solving problems from a slightly different perspective:
>
> 1. Migrating large OO applications into component-based ones,
> 2. Migrating Java legacy apps into micro-services and SOAs, and
> 3. Migrating Java legacy apps into aspect-oriented ones.
>
> All these problems require similar steps to migrate Java legacy apps to OO frameworks and plug-ins. One of the main reasons could be the urgent needs to migrate, based on refactorings, towards new emerging domains (since these domains did not exists more than 20 years ago) such as SOA, aspect-oriented systems, etc. That could be a nice example on how the refactoring "objectives" evolved and changed over time... (for example, nowadays the objectives of refactoring mobile apps are different than refactoring regular applications since there is much more emphasize on energy consumption, memory usage, etc.)"

We agree with these observations, but the lack of scripting tools really puts many of these objectives (and results) out of reach for most practitioners.

Works like [90, 34] acknowledge the need to restructure or refactor a program in order to generalize some part of the programs and make it reusable. However, they lacked support from an automated tool to apply all the code changes required to obtain a correct generalization.

Another representative paper, [35], proposes an approach to automatically identify a set of abstractions that could be made from program declarations. This research is motivated by "comprehension improvement through more abstract constructs, re-architecture of existing systems to improve their maintenance, or migration to new paradigms". Their approach is based on *software clustering*, which consists on identifying groups of program declarations that may correspond to more abstract artifacts. They applied their clustering approach to ten `C` programs. This works presents the formulation of their approach, however there is no evidence of any refactorings or transformations being applied.

Chapter 7

BttF Partitioning of Non-OO Programs (GAML) into SPLs

7.1 Introduction

Multi-Agent Systems (MAS) concepts and prototypes date back 20 years. Compared with Software Engineering, MAS is a rather young field. MAS aims to support future software-intensive societies by "enabling cooperation, coordination, and evolution of large-scale mixed human-machine systems" [61].

MAS are complex systems conformed by agents. An *agent* is a program and its key characteristic is autonomy, which it displays by taking actions over the environment within it is deployed. The actions are determined by an agent's goals [109, pp. 15, 16]. In a MAS, the main or global goal is accomplished in a distributed way, every agent contributes with its actions and interacts with other agents in order to achieve a system's goal [100]. A MAS is decentralized by nature, agents have partial knowledge and control over the environment they are deployed in [44].

MAS can be used to represent and investigate artificial and natural phenomena. They are used in a large range or research areas, like computer games, economics, health care, urban planning and social sciences. However, the development of agent platforms is usually driven by the domain of use each research community has. This has lead to numerous MAS platforms [52]. However, this diversity of MAS platforms has lead to a "lack of proven methodology and industrial-strength toolkits for creating agent-based systems" [40].

Among the research topics to advance MAS engineering, is the effort to combine MAS and SPL approaches: *Multi-Agent Systems Product Lines* (MAS-PL) [75]. Most of the research in MAS-PL is in methodological aspects, although there has also been some tool development. None of this work has tried to partition a MAS into features. In this case, we use BttF as a partitioning tool for a set of MAS programs, specifically MAS simulators developed in GAMA platform. A following section describe this platform and its programming language.

The motives that made us apply BttF as a partitioning tool in this case were:

- While working on developing simulators in *GIS*[1] *& Agent-based Modeling Architecture* (GAMA), it became evident that GAMA lacks high level reuse. Which, in our experience, is a common situation among MAS simulators development platforms.
- The way a MAS simulator is built tends to have a cross-cutting design. A functionality is weaved through the code of every modeled entity. We consider that this design is very agreeable with SPL concepts [11]. Each functionality added to a simulator needs to be reflected across entities. There is a clear mapping from implemented functionality and a SPL feature.

Later sections explain how we applied BttF in this case, the results we obtained, and current related work.

7.2 GAMA Platform and GAML Language

GAMA is an open source modeling and simulation platform. It is a "complete modeling and simulation environment for building rich spatially explicit agent-based simulations" [30].

GAMA development started in 2007; it is based on Eclipse and `Xtext`.[2] It provides tools to develop and experiment highly complex models by the integration of "agent-based programming, geographical data management, flexible visualization tools and multi-level representation". GAMA has been successfully used in several projects related to "environmental decision-support systems, urban design, water management, biological invasions, climate change adaptation or disaster mitigation" [42].

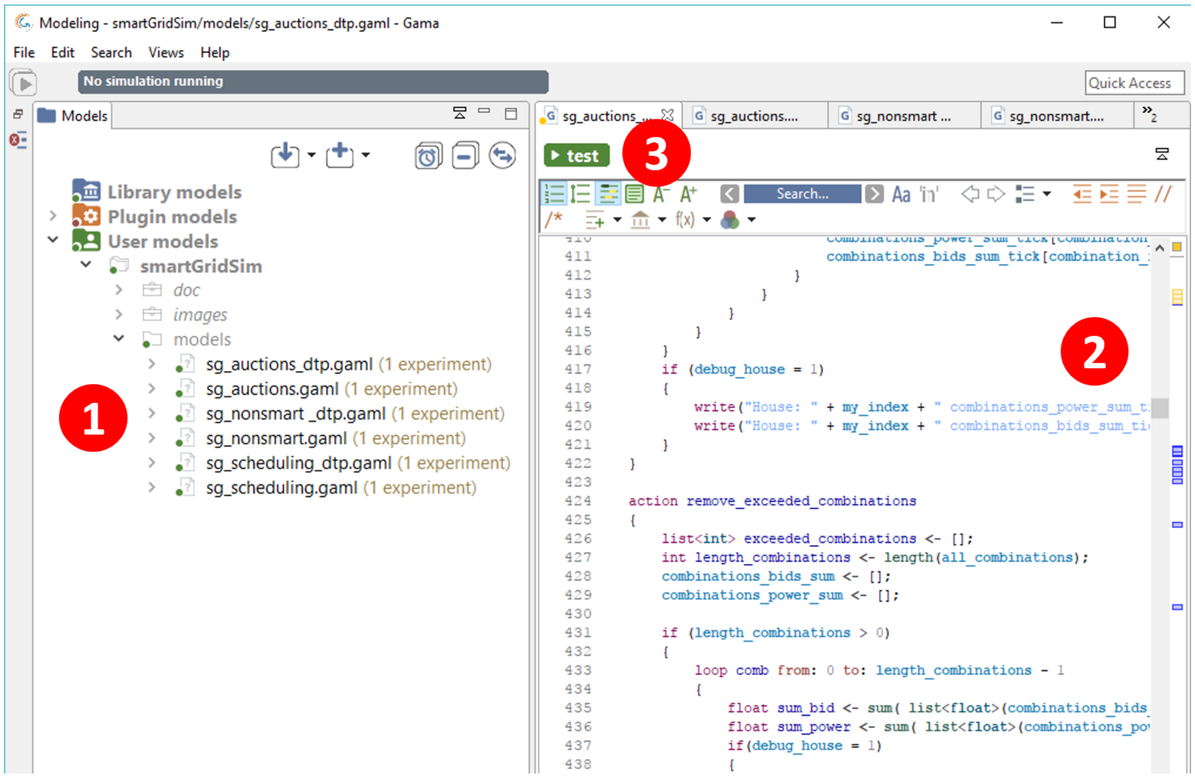
GAMA simulations are programmed in the *GAma Modeling Language* (GAML), which is an agent-oriented modeling language [42]. A GAML program is called a *model*, its containing file has the `gaml` extension. Figure 7.1 shows GAMA IDE; bubble 1 shows the list of models, bubble 2 shows the code editor, bubble 3 points to the available experiments configured for the model on screen, in this case, there is only one experiment named `test`.

Figure 7.2 shows a basic GAML model structure, which is composed of the following main declarations [77]:

- **Model Name**. A model starts with its name declaration.
- **Imports**. It is possible to import other models. Imports are optional.

[1] Geographic Information System.

[2] `Xtext` is an open-source framework for development of programming languages.`https://www.eclipse.org/Xtext/`


Modeling - smartGridSim/models/sg_auctions_dtp.gaml - Gama
File Edit Search Views Help
No simulation running
Quick Access
Models
Library models
Plugin models
User models
smartGridSim
doc
images
models
sg_auctions_dtp.gaml (1 experiment)
sg_auctions.gaml (1 experiment)
sg_nonsmart _dtp.gaml (1 experiment)
sg_nonsmart.gaml (1 experiment)
sg_scheduling_dtp.gaml (1 experiment)
sg_scheduling.gaml (1 experiment)
sg_auctions_...
sg_auctions....
sg_nonsmart ...
sg_nonsmart....
test
Search...
1
2
3

```
                            combinations_power_sum_tick[combination
                            combinations_bids_sum_tick[combination_
                        }
                    }
                }
            }
        }
        if (debug_house = 1)
        {
            write("House: " + my_index + " combinations_power_sum_t
            write("House: " + my_index + " combinations_bids_sum_ti
        }
    }

    action remove_exceeded_combinations
    {
        list<int> exceeded_combinations <- [];
        int length_combinations <- length(all_combinations);
        combinations_bids_sum <- [];
        combinations_power_sum <- [];

        if (length_combinations > 0)
        {
            loop comb from: 0 to: length_combinations - 1
            {
                float sum_bid <- sum( list<float>(combinations_bids_
                float sum_power <- sum( list<float>(combinations_pov
                if(debug_house = 1)
                {
```

Figure 7.1: GAMA IDE.

- **Global**. A `global` declaration contains attributes and methods declarations that are visible and modifiable from any part of the model. In GAML, methods have two flavors: `actions` and `reflexes`; an `action` is executed by an explicit call statement, like an OOP method; a `reflex` is executed automatically on each tick while the simulation is running.

- **Species**. A `specie` is used to define *types* of agents, it usually maps to an entity, for example, in a predator-prey model, a `specie` could be the predator and another `specie` the prey. Any number of instances of a `species` can be deployed in a simulation. A `specie` definition comprises attributes, `actions` and `reflexes`. A `specie` is similar to an OO class, it is encapsulated from the rest of the model segments, however its attributes are visible and modifiable from any other part of the model, and its `actions` can also be executed from any other part of the model.

- **Experiment**. An `experiment` contains the simulation parameters and the definition of the output. An output for example could be a graphical display. A model can contain any number of experiments. There are two types of `experiments`, `gui` or `batch`; `gui` experiments allows to display a graphical interface in which a user could set values for the input parameters; a `batch` experiment allows to execute numerous successive simulation runs.

```
model name_of_the_model

import "relative_path_to_a_model_file"

global {
    // definition of global attributes, actions, and reflexes
}

species my_specie {
    // definition of attributes, actions, and reflexes
}

experiment my_experiment type: gui //type is gui or batch
{
    // definition of input parameters
    output {
        // outputs definition
    }
}
```

Figure 7.2: The `@Feature` Annotation Type

7.3 Obtaining a CRG from a GAML Program

B$^{t}_{t}$F uses Eclipse's Java *Abstract Syntax Tree* (AST) support to obtain a CRG for Java programs. However, ASTs are not exclusive to Java programs. Programming languages developed with `Xtext` rely on *Eclipse Modeling Framework* (EMF).[3] Through EMF, a programming language created using `Xtext` can produce ASTs [110].

GAML was created using `Xtext`, therefore it is possible to obtain ASTs for the models or programs developed with this language. However, GAMA platform and IDE do not have an implemented option to traverse a GAML model's AST, and print the data we require to generated a PT's CRG. Therefore, we had to modify the source code of GAMA platform to implement a functionality that would traverse a model's AST and output what we required to compute a CRG.

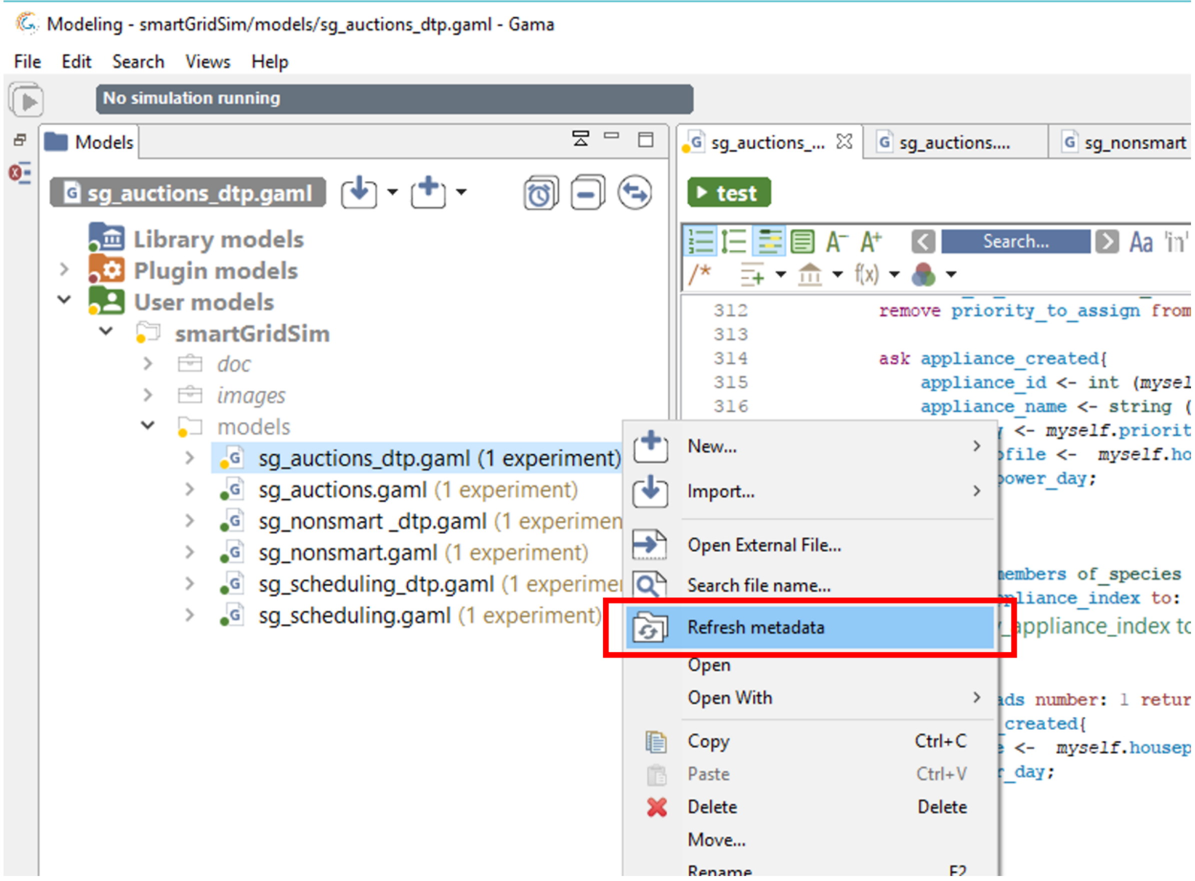
The GAMA platform is an open source project. Its source code in available in GitHub[4]. We cloned GAMA platform repository on August 2016, the version we obtained was a beta version of GAMA 1.7. After exploring and understanding the architecture and implementation of GAMA platform, we were able to implement a model's AST traversal method that would output a CSV file with the data required to generate a B$^{t}_{t}$F's CRG.

We added our implementation to an existing functionality. Figure 7.3 shows the contextual menu shown for a model, which has an option named `Refresh metadata`. We added in-

[3] Eclipse Modeling Framework supports the creation of building tools (i.e. programming languages and IDEs), `http://www.eclipse.org/modeling/emf/`

[4] GAMA platform source code repository is in `https://github.com/gama-platform/gama/`.

structions to the implementation of the method executed by this option. When `Refresh metadata` option is executed, the AST of the selected model is traversed, and BttF's CRG required data is obtained and saved in a CSV file (`CRGcsv`). The details about this modification and an example of a `CRGcsv` are contained in Appendix D.



```
        remove priority_to_assign from

        ask appliance_created{
            appliance_id <- int (mysel
            appliance_name <- string (
```

Figure 7.3: GAMA refresh metadata option.

A `CRGcsv` is basically a list of all the declarations' references found in a program, in this case, in a GAML model. Each line of the file contains a reference, these are the required attributes for a reference:

- Identifier of the declaration making a reference to another declaration, we named this `call_from`.
- Identifier of the declaration that is referenced, we named this `call_to`.
- `call_from` declaration type.
- `call_to` declaration type.

For GAML, we have these declaration types:

- `Declaration`. This is a field declaration.

- `Action`. This is a method that is executed by an explicit call in the model.
- `Reflex`. This also is a method, but is automatically executed on every tick of a simulation.
- `Specie`. Equivalent to a Java class. In can contain attributes and methods.

In Chapter 4 we mentioned the possibility of executing BttF as a standalone program (as opposite of its Eclipse plug-in execution mode). The standalone version of BttF allows a user to input a `CRGcsv`. This file is read by BttF, and parsed into a CRG. As part of this process, BttF maps declaration types to the types that are known to it: classes, methods, fields, etc.[5] Currently BttF is capable of performing this mapping for GAML and Java languages. As BttF was originally conceived to partition Java programs, the mapping process for this language is a natural fit. For GAML we have these declaration types mapping:

GAML declaration type	BttF **declaration type**
`Declaration`	Field
`Action`	Method
`Reflex`	Method
`Specie`	Class
`Experiment`	Class

Table 7.1: GAML declarations' types mapping to BttF's declaration types.

7.4 Experimental Setup

Like the experiments in previous chapters, expert knowledge is also required in order to partition a GAML model. For this evaluation we used six GAML models that we were part of their development. These models implement electrical smart grid related simulations using intelligent agents that participate in a combinatorial auction to buy power from the electrical grid [16].

Table 7.2 lists the programs selected for this evaluation.[6] Programs are sorted by `LOC` (Lines of Code).[7] `# of Features` is the number of features contained in a program's partitioning feature model. `# of Declarations` refers to the number of declarations found in a program's CRG.

[5] BttF's declaration types are described in **??**.

[6] The source code for these programs is available in `https://github.com/prianguloBttFTestProjects/tree/master/smartGridSim/models`

[7] Lines of code in this case are the number of lines contained in a model's file.

Program	**LOC**	**# of Features**	**# of Declarations**
`sg_scheduling_dtp`	1612	3	217
`sg_auctions_dtp`	1597	3	204
`sg_auctions`	1588	2	194
`sg_scheduling`	1559	2	207
`sg_nonsmart_dtp`	593	2	112
`sg_nonsmart`	544	2	101

Table 7.2: GAML programs to partition into SPLs with B^t_tF.

7.5 Research Questions

For this scenario of partitioning a GAML model into a SPL, we formulated one research question. We want to compare the inference rates obtained with this experiments versus the inference rates obtained with Java programs partitioned into SPL.

RQ1 How similar are the inference rates obtained when partitioning GAML models into SPLs, to the inference rates obtained when partitioning Java programs into SPLs?

To have a correct assessment, we need to compare each GAML model partitioning inference rate, with the average inference rate obtained in an equivalent set of experiments. For example, if a GAML model was partitioned using a feature model with configuration *base heavy*,[8] then we have to compare its obtained inference rate with the average inference rate obtained with the Java experiments that were partitioned with the same configuration.

7.6 Results

Every program used in this evaluation was partitioned into a SPL using B^t_tF. Four of the programs had two features in their feature model, the other two programs had three features in their feature model. For the programs with a two-feature feature model, we performed a single experiment. The results of these experiments are to be compared with the equivalent type of Java SPL partitioning experiments. For the programs with a three-feature feature model, we performed two experiments:

E1 We used the original three-feature feature model. We cannot compare these results with JAVA SPL partitioning experiments[9], therefore we defined the next type of experiment,

[8] Base heavy means most of the declarations are assigned to `BASE` feature.

[9] This comparison would lead to spurious results. For a correct comparison it is necessary that the set of experiments that are being compared, are performed on similar conditions.

E2 We combined the second and the last feature into a single one, aiming for a *top heavy* configuration[10].This way we could compare with Java E2 (top heavy) SPL partitioning experiments.

Table 7.3 contains the general results obtained with these experiments. The fields in this table are as follows:

- Type is the experiment type name:
  - Original. The original feature model was used. We used this type of experiment for sg_scheduling_dtp and sg_auctions_dtp programs, their feature model contained three features.
  - TH. A top heavy[11] configuration was used. We used this type of experiment for sg_scheduling_dtp and sg_auctions_dtp programs, we comprised their second and third feature into a single feature TOP.
  - Original (TH). The original feature model was used and it turned out to be a top heavy configuration. We used this type of experiment for programs with a two-feature feature model.
  - Original (BH). The original feature model was used and it turned out to be a base heavy[12] configuration. We used this type of experiment for programs with a two-feature feature model.
- N is the number of features in the partitioning feature model.
- F1 is the number of declarations contained in the first or BASE feature.
- F2 is the number of declarations contained in the second feature. For the experiments with a two-feature feature model, this feature was the last of TOP.
- F3 is the number of declarations contained in the third feature, if there was a third feature in the feature model.
- FP is the number of fprivate facts provided by an expert.
- FP% is the percentage of FP,

$$\text{FP\%} = \frac{\texttt{\# of fprivate facts}}{\texttt{\# of program declarations}} \cdot 100 \tag{7.1}$$

- I is the overall number of declarations that were inferred. For this scenario, B$^{t}_{t}$F makes five types of inferences. These are mentioned in later bullet points.

[10] A top heavy top configuration assigns most of the declarations to the TOP or last feature.

[11] Top heavy means most of the declarations were assigned to the TOP or last feature.

[12] Base heavy means most of the declarations were assigned to the BASE or first feature.

- `I%` is the percentage of declarations inferred by `BttF`,

$$\texttt{I\%} = \frac{\texttt{\# of BttF inferences}}{\texttt{\# of program declarations}} \cdot 100 \tag{7.2}$$

- `IF` is the number of declarations inferred because they reference a `fprivate` method or field.
- `IC` is the number of declarations inferred because they belong to a `fprivate` container.
- `IL` is the number of declarations that were assigned `fprivate` because they belong to the last feature in the feature model.
- `IO` is the number of declarations inferred because they had only one feature in bounds.
- `IN` is the number of declarations inferred because they have no references other than to its container.
- `H` is the number of identified `hook` methods.

Type	**N**	**F1**	**F2**	**F3**	**FP**	**FP%**	**I%**	**I**	**IF**	**IC**	**IL**	**IO**	**IN**	**H**
`sg_scheduling_dtp`														
Original	3	63	20	134	0	0%	21.2%	46	0	0	46	0	0	1
TH	2	62	155	-	0	0%	25.8%	56	0	0	56	0	0	0
`sg_auctions_dtp`														
Original	3	75	12	117	0	0%	32.8%	67	2	0	65	0	0	2
TH	2	73	131	-	0	0%	32.8%	67	0	0	67	0	0	0
`sg_auctions`														
Original (TH)	2	88	106	-	0	0%	26.3%	51	0	0	51	0	0	0
`sg_scheduling`														
Original (TH)	2	69	138	-	0	0%	22.7%	47	0	0	47	0	0	0
`sg_nonsmart_dtp`														
Original (BH)	2	109	3	-	0	0%	55.4%	62	0	0	0	62	0	0
`sg_nonsmart`														
Original (TH)	2	47	54	-	0	0%	33.6%	34	0	0	25	9	0	0

Table 7.3: GAML SPLs partitioning results.

7.6.1 RQ1. How similar are the inference rates obtained when partitioning GAML models into SPLs, to the inference rates obtained when partitioning Java programs into SPLs?

To answer this question, we need to separate the experiments in two groups:

- Top heavy experiments. Five of the programs were partitioned with a top heavy configuration. We can compare this group with Java `E2` (top heavy) SPL partitioning experiments, because they were partitioned using a similar configuration.

- Base heavy experiment. One of the experiments, `sg_nonsmart`, was partitioned using a feature model with a base heavy configuration. Let us refer to this experiment's result as `r`, we need to determine the probability of `r` of being *part* of the results obtained with Java `E3` (base heavy) SPL partitioning experiments. This means we need to evaluate the likelihood of `r` happening if we were partitioning a Java program using a base heavy configuration.

Top Heavy Experiments

To compare GAML top heavy experiments results (GTH-results) with Java top heavy experiments results (JTH-results), we need to perform a hypothesis test. A simple comparison of results' means is not enough, as the results come from different experiments. As mentioned in a previous chapter, *Hypothesis testing* is a process that allows to evaluate claims about a population. A hypothesis test uses data obtained from a sample to make a decision about whether a claim can be accepted or rejected [14, p. 400–430]. In this case we are claiming that GTH-results are equal than JTH-results. We claim this, because both types of experiments used the same type of feature model configuration for partitioning.

The formula to use on hypothesis testing depends on the attributes of the data that we have as a result from our experiments. We have two samples[13], one that represents GTH-results, and another that represents JTH-results.

Our samples are considered small, for GTH-results we have five values, and for JTH-results, seven. Given this, we need to use *t-test* as our statistical test [27, p.357, 358]. We compare the means of both samples. And our claim is that GTH-results mean is equal than JTH-results mean. The formal enunciation is:

$$\begin{aligned}
&H_0 : \mu = 64.67, && \text{this is the mean obtained with JTH-results,}\\
& && \text{we claim that this mean remains true for GTH-results}\\
&H_a : \mu \neq 64.67
\end{aligned}$$

To perform this hypothesis test we used R-Project, which is an environment for statistical computing [81]. The function we used is `t.test` from R's core package `stats` [82]. We provided this function with the values for H_0 (JTH-results) and H_a (GTH-results), and specified that we were conducting a *two-sided* comparison, which evaluates if H_a's mean μ is *different* than H_0's mean μ. By default, this test is performed with a confidence value (α) of 0.05%. This means that we have 95% certainty over the decision we make about rejecting (or not) our H_0 hypothesis.

[13] We say samples, because we did not conduct exhaustive experimentation, which would imply testing with every existing Java program, which is impossible.

The result obtained was:

$$p\text{-}value = \mathbf{0.0029} \tag{7.3}$$

The *p-value* (probability value) is the probability of obtaining $\mathtt{H_a}$ results when $\mathtt{H_0}$ is true. In this case, it is the probability of obtaining the mean stated in $\mathtt{H_a}$ when the true population mean is the one stated in $\mathtt{H_0}$. P-value measures the probability of $\mathtt{H_a}$ results being sufficiently evident and consistently better to disprove the results obtained with $\mathtt{H_0}$. To interpret a p-value we follow these rules [14, p. 418]:

$$\begin{array}{ll} p\text{-}value < \alpha, & \text{reject } \mathtt{H_0} \\ p\text{-}value \geq \alpha, & \text{do not reject } \mathtt{H_0} \end{array}$$

Since our p-value 0.0029 is less than our confidence value α 0.05, we have evidence to reject $\mathtt{H_0}$. This means that **we do not have evidence to support that the results obtained with GTH-results are similar or equivalent to the results obtained with JTH-results**.

For a reader comparing the inference rates obtained with JTH-results, and the inference rates obtained with GTH-results, it could be obvious that GTH-results were much lower than JTH-results.

The question now is why this difference in the inference rates? Both set of experiments were performed with a similar configuration. The difference in these results lies on the number of references among the declarations in the `TOP` layer. For Java programs, assigning a declaration to `TOP` feature had as a common effect *dragging* at least one more declaration to be part of the `TOP` feature[14]. For GAML programs we found that most of the declarations assigned to the `TOP` feature had references to the `BASE` feature, and very few of them would reference another declarations in the `TOP` feature. This situation reduced the impact that inferences for referencing a declaration in the last feature had. For GAML programs it was observed that declarations in the `TOP` feature had few references among each other.

Base Heavy Experiment

For this case we have a single base heavy GAML experiment (GBH-result). We need to obtain the probability of having this result if we were performing base heavy Java experiments (JBH-results). This posses a challenge, we do not know all the possible results that could be obtained when partitioning a program using a base heavy configuration[15], therefore, we need a way of measuring how likely is for GBH-result to be a value of a distribution that represents JBH-results.

To obtain this probability we performed two steps:

[14] For the Java programs we used for evaluation in SPL partitioning, it was rare to find isolated declarations, or declarations with few references. It could be said that Java programs CRG was a dense graph[105], it had more edges than vertices. When a declaration `d` is assigned to the `TOP` feature, it is set as `fprivate`, therefore, declarations that reference `d` are subject to be assigned to `TOP` feature too.

[15] To obtain all the possible results we would have to partition every possible Java program and every possible GAML program or model, which is impossible.

Step 1. Obtain JBH-results distribution. Since JBH-results is composed of a small number of values, it is likely to pass a *normality test*, which asserts that a sample has a normal distribution. The adequate test for this case is *Shapiro-Wilk test for normality*. This normality test applies to small samples, of size 50 or less[83].

To perform this normality test we used R-Project, which is an environment for statistical computing [81]. The function we used is `shapiro.test` from R's core package `stats` [82]. We provided this function with the values for JBH-results. By default, this test is performed with a confidence value (α) of 0.05%. This means that we have 95% certainty over the normality of the sample.

The result obtained was:

$$p\text{-}value = \mathbf{0.777} \tag{7.4}$$

For `shapiro.test`, this p-value is interpreted as follows:

$$\begin{array}{ll} p\text{-}value < \alpha, & \text{do not passed normality test} \\ p\text{-}value \geq \alpha, & \text{passed normality test} \end{array}$$

Since 0.777 is greater than 0.05 (α), we can assert that JBH-results appears to have a normal distribution. Since our sample passed the normality test, we can proceed to the next step[16].

Step 2. Given that JBH-results follows a normal distribution, now we have to determine what is the probability of GBH-result to be part of this distribution. In this case we need to perform a statistical test that evaluates the probability of GBH-result to be inside the distribution curve of JBH-results. The formula and procedure to perform this were taken from [14, p. 427-429]:

- Convert GBH-result into a *t-value*, this is similar to normalizing a value, the formula for this is:

$$\mathtt{t} = \frac{\overline{\mathtt{X}} - \mu}{\mathtt{s}/\sqrt{\mathtt{n}}}$$

where,

$\overline{\mathtt{X}}$ = Observed value to convert, in this case GBH-result

μ = Mean of the values in JBH-results

$\mathtt{s}$ = Standard deviation of the values in JBH -results

$\mathtt{n}$ = Number of observations in JBH-results

[16] If our sample wouldn't have passed the normality test, we would have had to perform other statistical tests to determine which statistical distribution our sample has.

- Obtain the p-value, this is the probability of our t-score to be inside the desired distribution curve. This is the formula =

  $$\mathtt{p-value} = 2 * \mathtt{pt(-abs(t),df)}$$

  where,

  `pt` = Is the distribution function for *t* statistic, it returns a p-value.

  `abs(t)` = Absolute value of `t`, which we obtained in the previous step

  `df` = Degrees of freedom, $\mathtt{n}-1$

  This formula makes use `pt`, which is the distribution function for *t statistic.* This function provides the probability of a $\mathtt{t}-\mathtt{value}$ to be under the `t` distribution curve. A reader might notice that this formula multiplies by 2 the probability obtained, this is because the formula considers both tails of the curve.

We executed these steps in R-Project, which is an environment for statistical computing [81]. For `pt` function, we used R's core package `stats`, which has a `pt` implementation [82]. We provided this function with the required values, as stated in the formula. The result obtained was:

$$\textit{p-value} = \mathbf{0.00028} \tag{7.5}$$

By default α in this case in 0.05, therefore our confidence interval is 95%. The p-value obtained is interpreted as follows:

$$\begin{array}{ll} \textit{p-value} < \alpha, & \mathtt{t} \text{ does not belong to that distribution curve} \\ \textit{p-value} \geq \alpha, & \mathtt{t} \text{ belongs to that distribution curve} \end{array}$$

Our p-value (0.00028) is less than our α (0.05), this means that there is no evidence, or support of GBH-result belongs to the same distribution curve that JBH-results have. Therefore, GBH-result is not similar to JBH-results.

As a reader might observe, the inference rated obtained with GBH-result (55.4%) is higher than the inference rates obtained with JBH-results, (37.9% was the highest obtained). Table 7.3 shows that actually the highest inference rate obtained with GAML experiments was precisely with this experiment (`sg_nonsmart_dtp`) model. For Java SPL partitioning experiments, `E3` (base heavy) experiments had the worst inference rate, as opposite to what we observed for GAML. For this experiment, all the inferences made were because there was only one feature in bounds, this kind of inference was rare in Java SPL partitioning. It could be said that for GAML programs, B$^{t}_{t}$F has a higher inference rate when the partitioning is made with a base heavy configuration, however, we only have one example of this behavior. More experiments would have to be performed to assert the validity of this affirmation.

7.7 Related Work

Many papers have been written on MAS product lines. Keep in mind that there is no tool or prior work directly comparable to `BttF` in migrating legacy MAS systems to MAS product lines. The research we cite in this section are contributions on MAS-PL development methodologies. They focus on the modeling phase when developing a MAS-PL from scratch, not on partitioning an existing MAS into a MAS-PL, which is the focus of `BttF`. Their contributions are useful for the analysis and design phases of a MAS-PL.

There are three main MAS-PL methodologies that support the modeling of a MAS-PLs:

- **MAS PASSI + SPL PLUS.** [54] first proposed a generative approach to develop MAS systems using aspect oriented techniques. In their proposed process they support MAS design using these modeling languages: `Agent−DSL`[53] and `aSideML`[19]. Both specifications are used for modeling variabilities and commonalities. In this work `AspectJ`[17] was used to support the implementation of their approach.

  Later, to overcome limitations of existing MAS modeling languages with regards to express variability (as required when modeling a SPL), [66, 65] proposed an agent variability modeling language that extended PLUS (Product Line UML-based Software Engineering) notation language [41] with the MAS PASSI (Process for Agent Societies Specification and Implementation) [22] design methodology. This research work details the challenges of correctly modeling commonalities and variabilities for a MAS-PL, which is critical for defining any SPL. They also describe specific problems that need to be addressed on MAS, that do not exist when designing a SPL for OO systems.

- **Gaia-PL.** [25, 26] proposed and created Gaia-PL methodology, which extends MAS Gaia methodology[111] to document variability per agent roles[18]. Gaia is the most commonly used methodology to guide the process of developing a MAS[18].

- **MaCMAS.** [73, 74] presents the MaCMAS methodology, and describes its application in a case study where a MAS-PL is developed. This methodology supports modeling complex MAS and focuses on multi-role interactions, interactions among agents.

Some of these papers presented examples of a product line derivation; all used AOP[19]as their derivation technique.

[17] `AspectJ` is Eclipse's aspect oriented implementation for Java programs [80].

[18] A MAS is usually designed around the agents that are to be developed and the role they will play in the system.

[19] *Aspect oriented programming* (AOP) is a paradigm that adds modularity to a program without modifying the source code structure, by adding instructions that allow the separation of cross-cutting concerns [104].

Chapter 8

Conclusions and Future Work

Partitioning a program, into an SPL or into a framework and plug-in, is hard. If performed manually, it is not only tedious, but prone to errors. The consistently good results we obtained with B^t_tF shows its usefulness as a tool to support the correct feature partitioning of a program, and therefore, we consider that it is the right path to reduce the adoption barrier SPL technologies currently have.

We successfully applied B^t_tF in three different scenarios, with results never obtained before with related research:

- **Partitioning of a Java program into an SPL**. This was our main evaluation scenario; B^t_tF was designed with this scenario on mind. For the best set of experiments we obtained an average inference rate of 64.7%; the highest inference rate obtained with these experiments was 84.5%, this means that for that specific experiment, a user only did 15.5% of the assignment work, and B^t_tF did the rest.

- **Partitioning of a Java program into a framework and plug-in**. For this scenario, B^t_tF's average inference rate was lower than for its main scenario, 39.5%. However, we integrated two state-of-the-art tools, B^t_tF and R3, to perform an almost fully automated[1] transformation of a Java program into a framework and plug-in structure. This kind of work has never been performed before.

- **Partitioning of a GAML programs into an SPL**. MAS development is currently an active research area. This has lead to the creation of numerous MAS development platforms [52], however, these platforms are not yet as mature as OO development platforms. One of the topics that recently got attention is the usage of SPL technologies on MAS, most of this research covers methodological challenges. B^t_tF is the first tool used to identify features in MAS programs. The best inference rate obtained with this scenario evaluation was 55.4%.

[1] Only those methods identified by B^t_tF as `hooks` were not transformed, this remains as a task that needs to be performed manually by developers. However, we found that `hook` methods tend to be rare; the maximum number of `hook` methods we identified with our experiments, was nine.

Related work that aim to extract features from a program [48, 76, 99] do not fully partition a program into features; instead, they extract one feature at a time, or a set of features that do not fully cover a program's codebase. This forces their evaluations to focus on the precision and recall of their results. In our case, our tool fully partitions a program's codebase, and our CRG-based approach asserts the correctness of the partitioning, therefore precision and recall are not relevant. Our focus was to reduce the amount of work a developer needs to perform. This is the reason why our main success metric is inference rate.

8.1 Future Work

We envision three additions we consider would improve `BttF`:

- Add a pre-partitioning step to help a developer get better acquainted with a program to partition. Even if a developer is an expert on a program codebase, s/he may struggle to remember the dependencies and affiliations of every declaration. We personally experienced this situation. We consider adding an extra tool to run searches over a program's source code; this tool would allow to search for keywords, naming conventions and patterns, to help group declarations in a way that would aid feature assignment.
- Allow a user to provide assignments, not only by following our designed assignment main path[2] , but also by order of the amount of declarations that reference a declaration. This way a developer could assign first those declarations that would have a larger impact in the partitioning process, even more if the declaration being assigned is `fprivate`.
- The analysis of `BttF` is based on a notion of time: that feature α was built before β, γ was built after β, etc. This information is available in *Version Control Systems* (VCSs). If the VCS repository of program could be mined, we could gather a tremendous amount of information – like when features were developed, and the coincidence of the introduction of declarations at that time. This is often the information that `BttF` asks of its users and that users have a hard time to provide. This, we believe, may have a great impact on raising `BttF` inference rates in the future.

With these additions, we believe `BttF` could be used to partition even larger programs, and ideally, commercial programs that are in need to adopt SPL technologies.

[2] `BttF`'s main path is described in Section 3.2.7

Appendix A

Refactorings versus Transformations

A refactoring script starts with a program s and applies a series of refactorings to produce a restructured program e. At each step of the refactoring process, preconditions are evaluated to verify the legality of that refactoring; if any precondition fails, the refactoring fails, and so too does the script. Figure A.1 illustrates how every refactoring r applied to a program s yields a program that is semantically equivalent to s.

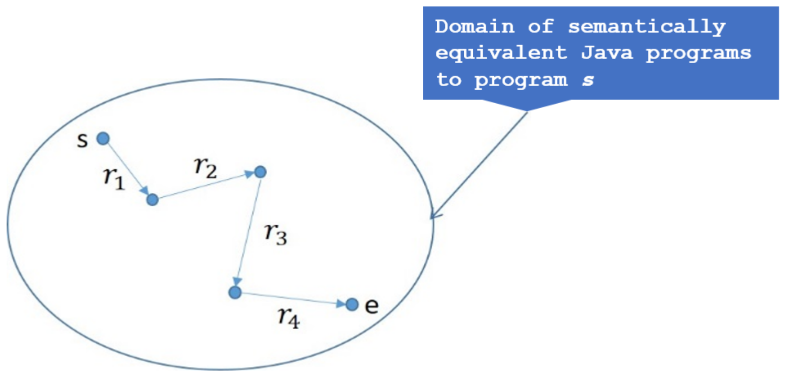

Domain of semantically equivalent Java programs to program *s*
s
r_1
r_2
r_3
r_4
e

Figure A.1: Refactoring process

Another approach to restructure a program s into e, is a *transformation script*, which is less restrictive. The only guarantee is that given input program s, the end program e will be behaviorally equivalent to s. The steps that are taken are transformations that do not necessarily preserve program behavior. This approach is similar to a database transaction, which preserves database consistency at the end of the transaction, but no guarantees are offered for any state of the database during the execution of the transaction. Consider Figure A.2, we start at program s. We apply transformations t_1, t_2, t_3, t_4 in this order to produce end program e. Note that the intermediate programs (denoted by red dots) are not behaviorally equivalent to s. It is even possible that these programs will not even compile. It is only the end result, program e, in which we are interested.

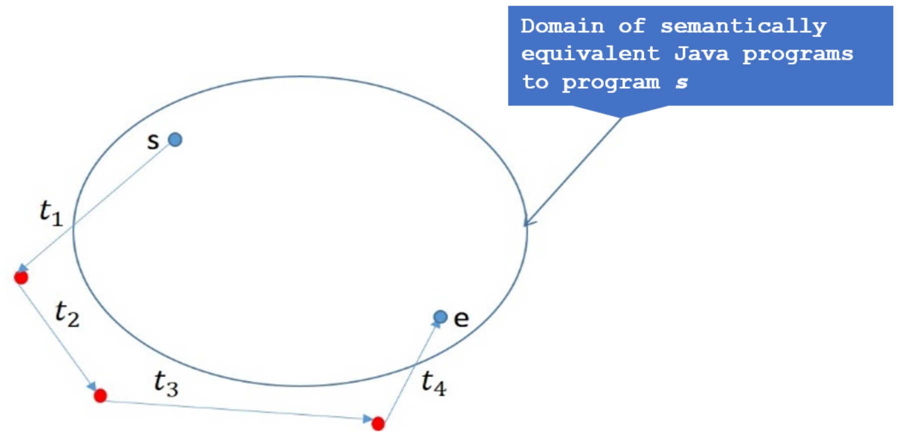

Domain of semantically equivalent Java programs to program *s*
s
e
t_1
t_2
t_3
t_4

Figure A.2: Transformation process

This second approach may seem odd, as one would like to guarantee the semantic equivalence of `s` and `e`. A refactoring engine on its own could perform transformations (not refactorings), but not guarantee that program `e` is behaviorally equivalent to program `s`. However, in this case, we have BttF which analyzes `s`, and plots the sequence of transformations to map `s` to `e`. We do not need to use refactorings (with their precondition checks) because, in effect, BttF's analysis is taking care of this. There is no need for the refactoring engine to duplicate (or complicate or verify) what BttF has already done. This approach is well-known, for example, in taking a single-threaded legacy application and rewriting its code into a multi-threaded application. The refactoring engine is really being used as a transformation engine. The tool that does the single-threaded-to-multi-threaded mapping performs the required analysis .

BttF supports this mapping process from `s` to `e`. BttF applies a set of rules to correctly determine the new location of declarations in the new structure of program `e`, and only needs a transformation engine to accomplish its code restructuring.

Appendix B

R3 framework and plug-in transformation script

```java
package r3forBttf;

import java.util.ArrayList;
import java.util.stream.Collectors;
import bttf.Element;
import bttf.ElementType;
import bttf.FWPlBelongLevel;
import bttf.Partition;
import bttf.PartitionHelper;
import gui.GuiConstants;
import gui.InputFile;
import p.actions.*;
import java.awt.Frame;

public class RefactoringScript {
 Partition partition;
 ArrayList<String> notFound = new ArrayList<String>();
 ArrayList<String> lifted = new ArrayList<String>();
 private String start_path = System.getProperty("user.home") + "\\Desktop\\BttF";

 public void applyBttFrefactorings() {
  InputFile inputFile = new InputFile();
  Frame parentWindow = new Frame("Bttf-R3");

  String file_name = InputFile.getFileFromFileDialog(parentWindow, start_path,
      GuiConstants.CSV_EXTENSION, "");
  this.partition = inputFile.get_elements_from_csv_file_nocheck(file_name);

  // FW is the list of plugin classes and interfaces that will belong to the
      framework
  ArrayList<RType> FW  = new ArrayList<RType>();

  ArrayList<Element> packages = (ArrayList<Element>)
      partition.get_elements().stream()
    .filter(e -> e.getElement_type().equals(ElementType.ELEM_TYPE_PACKAGE))
```

```
   .collect(Collectors.toList());

 for(Element packag : packages){
  RPackage original = RProject.getRPackage(packag.getIdentifier());

  RPackage frame = null; // framework package
  RPackage plugn = null; // plugin package
  RClass abstractFactory = null; // (abstract) factory class of the framework
  RClass concreteFactory = null; // (concrete) factory class of the plugin

  // Step 1: create framework package and rename original
  RPackage parent = original.getParent(); //get parent package
  String origName = original.getName();
  String pluginName = origName+"_plugin";
  String frameName = origName+"_framework";

  original.rename(pluginName);
  frame = parent.makePackage(frameName);
  plugn = original;

  // Step 2: create an empty framework abstract factory class and empty
  // plugin concrete factory class that have a subclassing relationship
  abstractFactory = frame.makeAbstractClass("Factory");
  concreteFactory = plugn.makeSubClass("Factory", abstractFactory);

  // Step 3: populate FW with original/plugin classes
  for (RPackageMember pm : plugn.getMembers()) { //pm could be a class, interface,
      enum, @Interface, package

   Element pm_bttf = getBttFElementFromR3PM(pm, origName, pluginName, frameName);

   if(pm_bttf != null && pm_bttf.getIdentifier() != null){
    boolean fully_belongs_to_framework =
        pm_bttf.belongLevelFW().equals(FWPlBelongLevel.FULLY_BELONGS_FW);
    boolean only_parts_belong_to_framework =
        pm_bttf.belongLevelFW().equals(FWPlBelongLevel.PARTIALLY_BELONGS_FW);


    // Step 3.a: if members of a FW concrete class have constructor calls,
    // the class must be processed as it partially belongs to the framework
    // so we can apply liftConstructorCalls later
    if((fully_belongs_to_framework == true)
     && !pm.isInterface()
     && pm.isConcrete()
     && pm_bttf.needsLocalConstructor()
     ){
      fully_belongs_to_framework = false;
      only_parts_belong_to_framework = true;
     }

    // Step 3.b: create abstract classes for classes that partially
    // belongs to the framework
    if (only_parts_belong_to_framework == true) {
     if (!pm.isInterface()) {
      System.out.println("make FW abstract class: " + pm.getName());
      RClass ac = frame.makeAbstractClass(pm.getName());

      if(pm.getExtends() != null){
       System.out.println("current extends of pm: " + pm.fullName + " " +
           pm.getExtends().fullName);
      }
      if(ac.getExtends() != null){
       System.out.println("current extends of ac: " + ac.fullName + " " +
           ac.getExtends().fullName);
      }

      ac.makeExtends(pm.getExtends());
      pm.makeExtends(ac);
      FW.add((RType)pm);
     }
     else if (pm.isInterface()) {
        RInterface i = frame.makeInterface(pm.getName());
        i.makeExtends(pm.getExtends());
      pm.makeExtends(i);
      FW.add((RType)pm);
     }
     else if (pm instanceof REnum || pm instanceof RAnnotation) {
      //BTTF DOES NOT ALLOW THIS because @interface and enum cannot extend
     }
    }
   }//END OF  if(pm_bttf != null && pm_bttf.getIdentifier() != null){


   // Step 4: lift constructor calls
   for (RType t : FW) {
    for(RTypeMember m : t.getMembers()){
     Element m_bttf = getBttFElementFromR3PM(m, origName, pluginName, frameName);
     if(m_bttf != null && m_bttf.getIdentifier() != null){
      if(!m_bttf.isIs_hook()){
       m.liftConstructorCalls(FW); //this method does the hard work
      }
     }
    }
   }
  }//END OF  for (RPackageMember pm : plugn.getMembers()) {


  // Step 5: move or pull apart types that partially belong to the framework
  for (RPackageMember pm : plugn.getMembers()) {
   Element pm_bttf = getBttFElementFromR3PM(pm, origName, pluginName, frameName);
   if(pm_bttf != null && pm_bttf.getIdentifier() != null){
    boolean fully_belongs_to_framework =
        pm_bttf.belongLevelFW().equals(FWPlBelongLevel.FULLY_BELONGS_FW);
    boolean only_parts_belong_to_framework =
        pm_bttf.belongLevelFW().equals(FWPlBelongLevel.PARTIALLY_BELONGS_FW);

    if((fully_belongs_to_framework == true)
     && !pm.isInterface()
     && pm.isConcrete()
     && pm_bttf.needsLocalConstructor()
    ){
     fully_belongs_to_framework = false;
     only_parts_belong_to_framework = true;
    }


    if(fully_belongs_to_framework == true){
     lifted.add(pm.getFullName());
     pm.liftAllTypes(FW);

     //move all contents to framework package
     pm.move(frame);
     pm.setPublic();
     for(RTypeMember m : pm.getMembers()){
      if(m.isConstructor()){
        m.setPublic();
        }
      else if(!m.isProtected()){
       m.setPublic();
      }
     }
     continue;
    }

    if (only_parts_belong_to_framework == true){
     RType absT = pm.getParent(); //created previously
     for(RTypeMember m : pm.getMembers()){
      Element m_bttf = getBttFElementFromR3PM(m, origName, pluginName, frameName);
      if(m_bttf != null && m_bttf.getIdentifier() != null){
       boolean m_fully_belongs_to_framework =
           m_bttf.belongLevelFW().equals(FWPlBelongLevel.FULLY_BELONGS_FW);
       if(m_fully_belongs_to_framework){
        if(!m.isConstructor() && ( m_bttf.isIs_hook() || m.isLocalFactory()) ){
        if(!m.isPrivate() && !m.isProtected() && !m.isPublic()){
         m.setProtected();
        }
        absT.makeAbstractMethodWithLiftedSignatureTypes(m,FW);
        }
         else if(m.isConstructor()){
           m.setPublic();
           RMethod f = concreteFactory.makeFactoryWithLiftedSignatureTypes(m, FW);
           abstractFactory.makeAbstractMethod(f);
           lifted.add(pm.getFullName());
           abstractFactory.liftAllTypes(FW);

           absT.addConstructor((RMethod)m);
         }
         else {
           lifted.add(m.getFullName());
           m.liftAllTypes(FW);
           m.promote();
           if(!m.isProtected()){
            m.setPublic();
           }
          }
       }//END OF  if(m.fully_belongs_to_framework){
       else{
        lifted.add(m.getFullName());
        m.liftAllTypes(FW);
       }
      }//END OF  if(m_bttf != null && m_bttf.getIdentifier() != null){
     }//END OF  for(RTypeMember m : pm.getMembers()){
     if(absT != null){
      lifted.add(absT.getFullName());
      absT.liftAllTypes(FW);
     }
     pm.liftAllTypes(FW);
    }//END OF  if (pm.only_parts_belong_to_framework == true){


   }//END OF  if(pm_bttf != null && pm_bttf.getIdentifier() != null){
  }//END OF  for (RPackageMember pm : plugn.getMembers()) {

 }//END OF for(Element packag : packages){
 printNotFound();
 printLifted();
}

private Element getBttFElementFromR3PM(RPackageMember pm, String origPName, String
    plPName, String fwPName){
 Element pm_bttf = null;
 String fullName = null;

 if (pm instanceof RType){
  fullName = ((RType)pm).fullName;
 }
 else if (pm instanceof REnum){
  fullName = ((REnum)pm).fullName;
 }
 else if (pm instanceof RAnnotation){
  fullName = ((RAnnotation)pm).fullName;
 }
 else{
  fullName = pm.fullName;
 }

 if(fullName.startsWith(plPName + ".") && !fullName.startsWith(origPName + ".")){
  fullName = fullName.replaceFirst(plPName + ".", origPName + ".");
 }
 pm_bttf = PartitionHelper.get_element_from_string(partition.get_elements(),
    fullName);

 if(pm_bttf == null){
  addToNotFound(fullName);
 }
 return pm_bttf;
```

```

}

private Element getBttFElementFromR3PM(RTypeMember tm, String origPName, String
    plPName, String fwPName){
 String fullName = null;

 if (tm instanceof RMethod){
  fullName = ((RMethod)tm).getSignature();
  fullName = fullName.replace("\\n\\r", " ");
  fullName = fullName.replace("\\r\\n", " ");
  fullName = fullName.replace("\\r", " ");
  fullName = fullName.replace("\\t", " ");
  if(fullName.contains(" throws ")){
   int endIndex = fullName.lastIndexOf(")", fullName.indexOf(" throws "));
   fullName = fullName.substring(0, endIndex+1);
  }

  fullName = fullName.replaceAll("(\\s+\\()", "(");
  fullName = fullName.replaceAll("(\\(\\s+)", "(");
  fullName = fullName.replaceAll("(\\s+\\))", ")");
  fullName = fullName.replaceAll("(\\s+,\\s{0})", ", ");
  fullName = fullName.replaceAll("(\\s){2,}", " ");

  fullName = fullName.trim();

  String parentFullName = ((RMethod)tm).getRType().fullName;
  if(!fullName.startsWith(parentFullName)){
   fullName = parentFullName + "." + fullName;
  }
 }
 else if (tm instanceof RFieldDeclaration){
  fullName = ((RFieldDeclaration)tm).getFullName();
  String parentFullName = ((RFieldDeclaration)tm).getRType().fullName;
  if(!fullName.contains(parentFullName)){
   fullName = parentFullName + "." + fullName;
  }
 }
 else{
  fullName = tm.getFullName();
 }

 if(fullName.startsWith(plPName + ".") && !fullName.startsWith(origPName + ".")){
  fullName = fullName.replaceFirst(plPName + ".", origPName + ".");
 }
 Element tm_bttf =
     PartitionHelper.get_element_from_string(partition.get_elements(), fullName);
 if(tm_bttf == null){
  addToNotFound(fullName);
 }
 return tm_bttf;
}

private void addToNotFound(String fullName){
 if(!notFound.contains(fullName)){
  notFound.add(fullName);
 }
}

private void printNotFound(){
 for(String s : notFound){
  System.out.println("Not found in BttF: " + s);
 }
}

private void printLifted(){
 for(String s : lifted){
  System.out.println("liftAllTypes(): " + s);
 }
}

}
```

Appendix C

Class diagrams of framework and plug-in programs

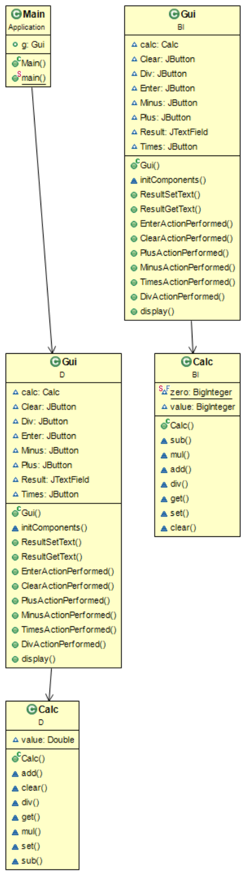


Figure C.1: `Calc` Java Program. Original class diagram.

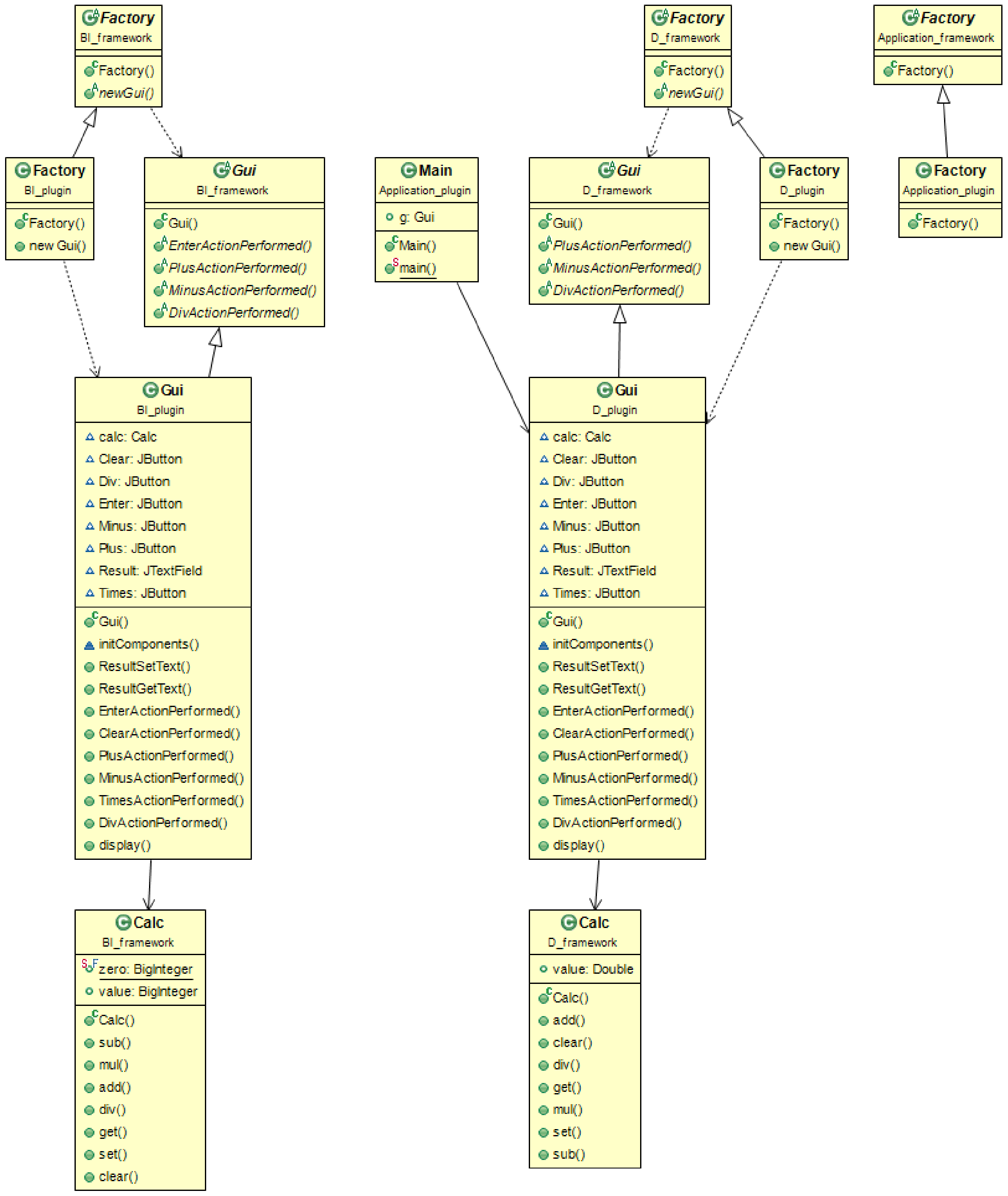


Figure C.2: `Calc` Java Program. Class diagram after framework and plug-in transformation.
]]>

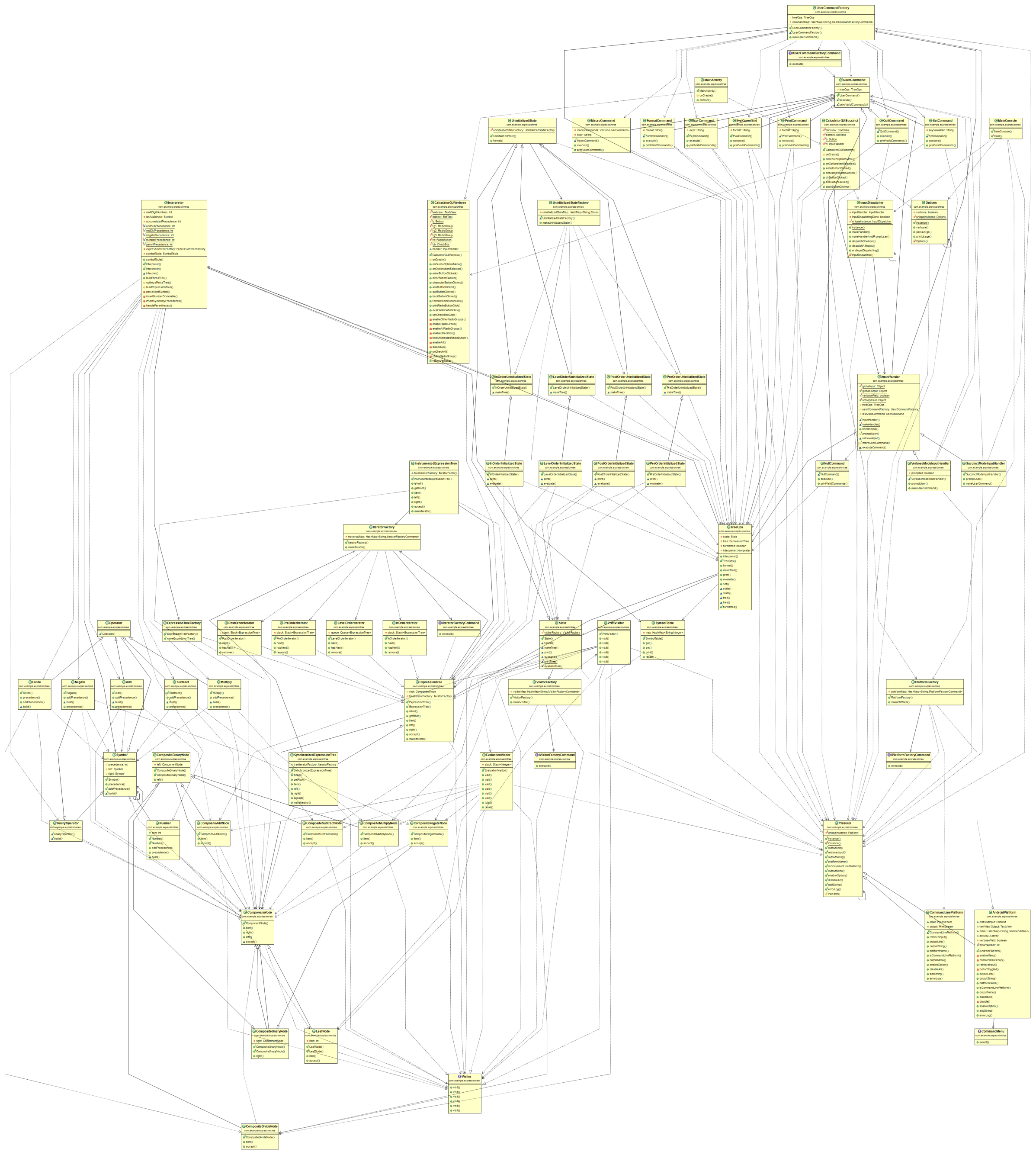


Figure C.3: `ExpressionTree` Java Program. Original class diagram.

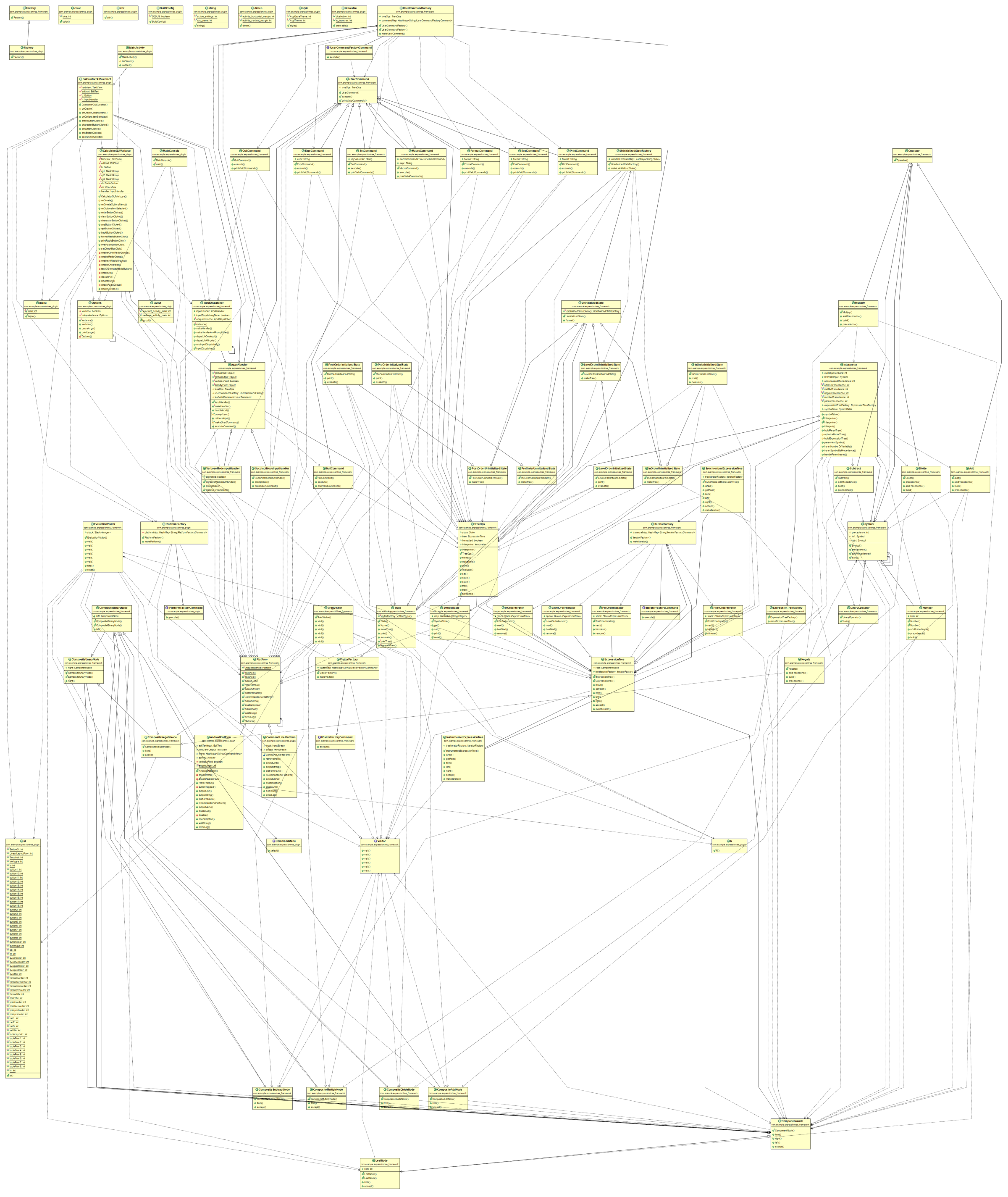


Figure C.4: `ExpressionTree` Java Program. Class diagram after framework and plug-in transformation.

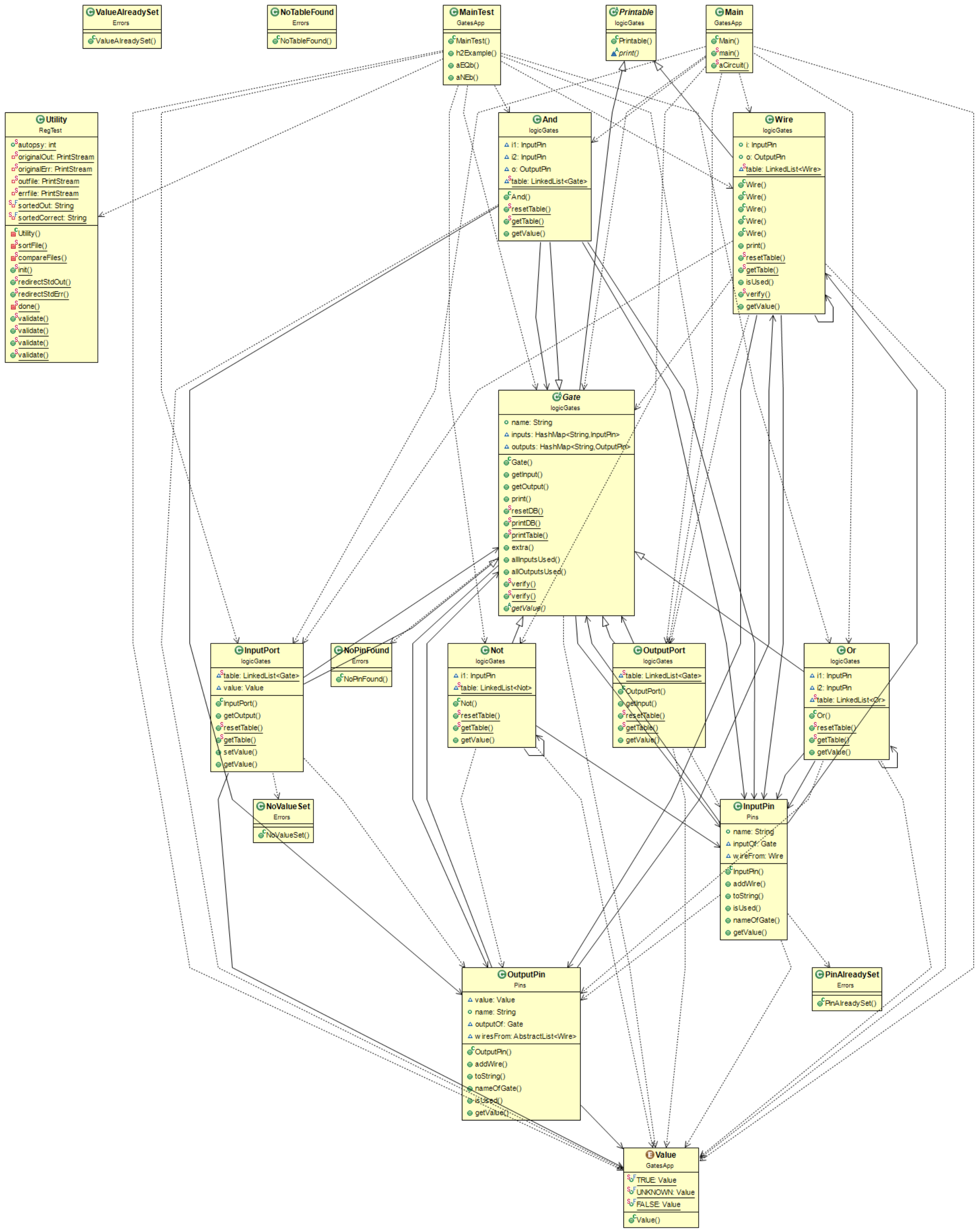


Figure C.5: `Gates` Java Program. Original class diagram.

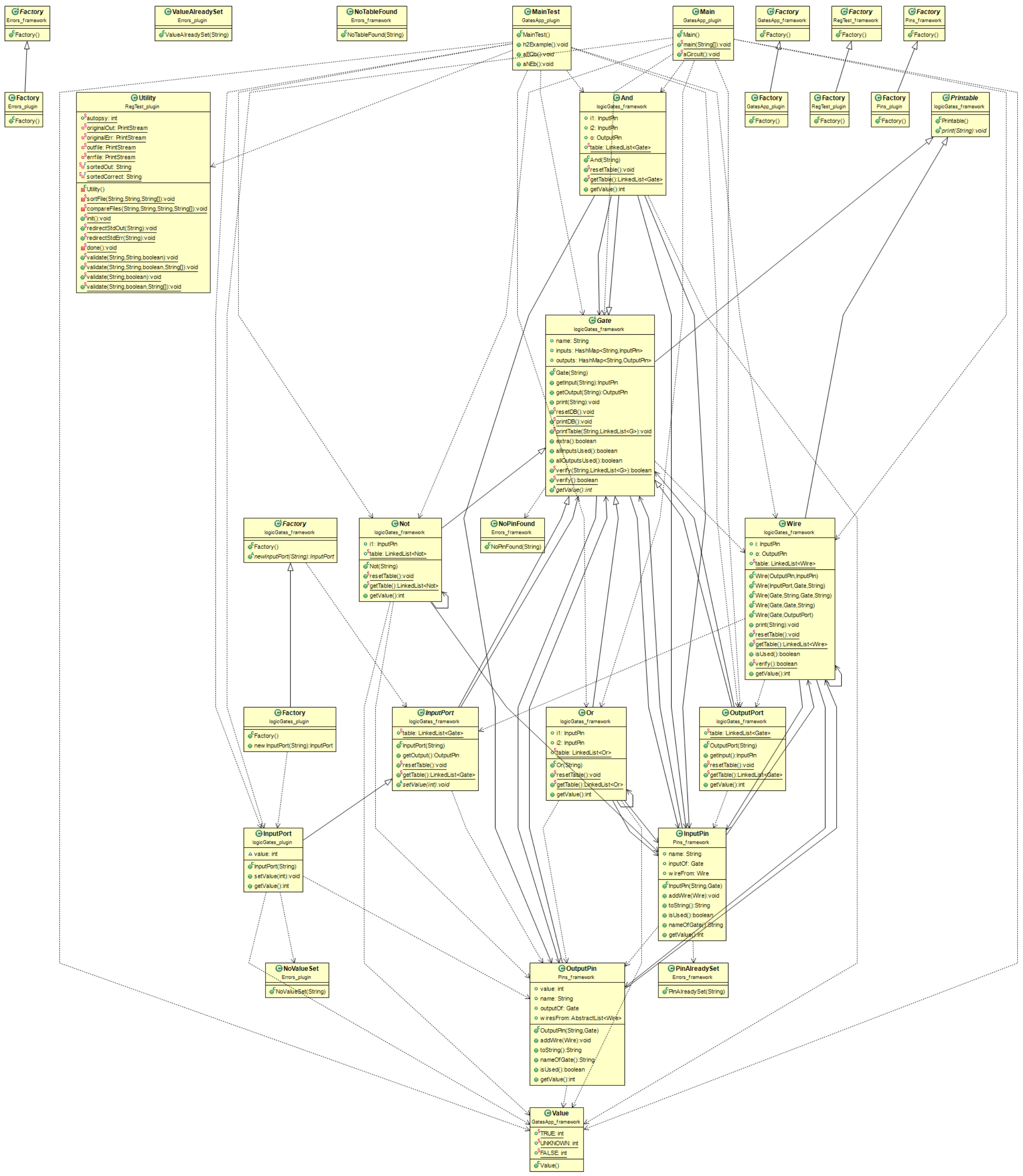


Figure C.6: `Gates` Java Program. Class diagram after framework and plug-in transformation.

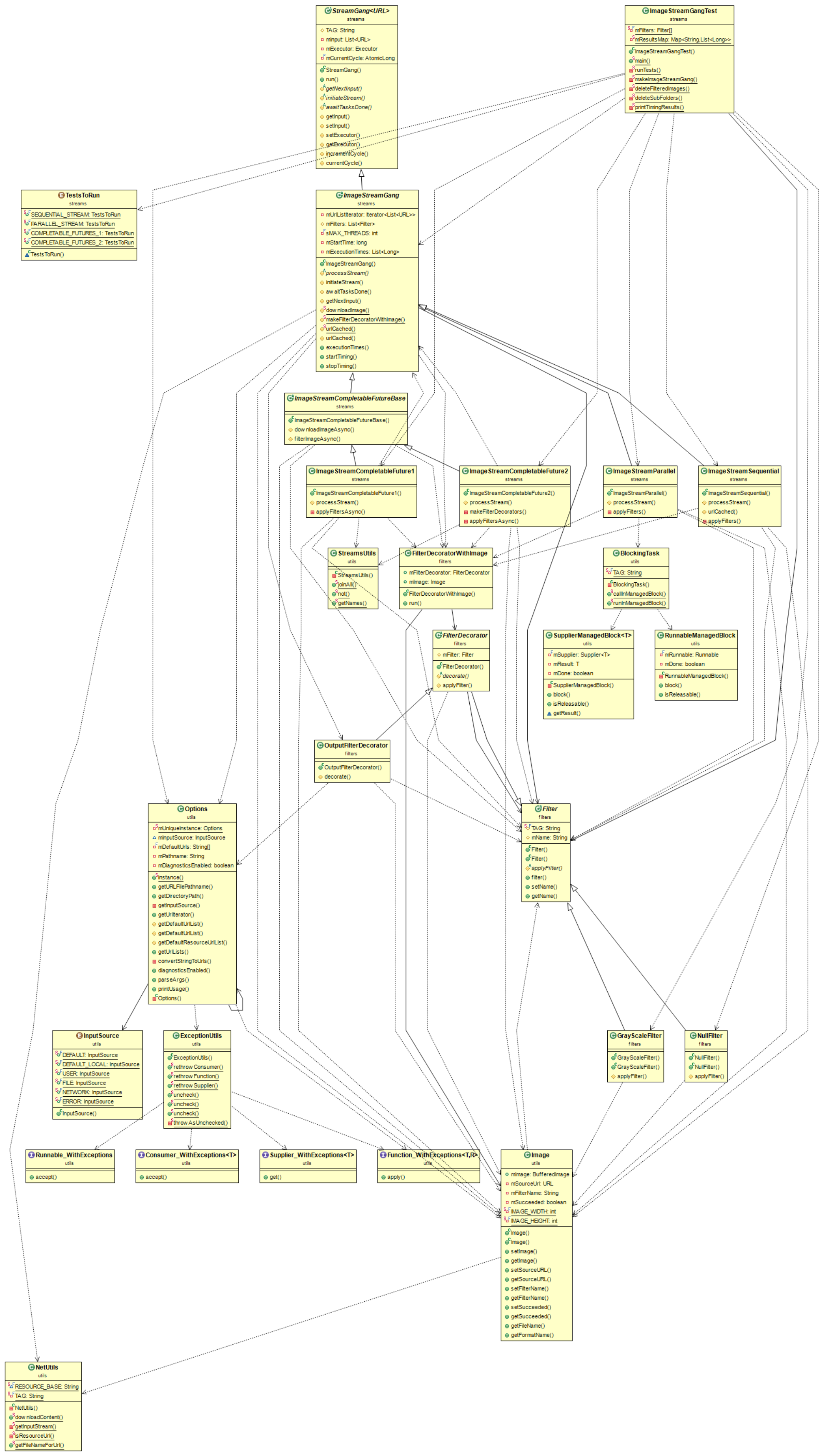


Figure C.7: `ImageStreamGangApp` Java Program. Original class diagram.

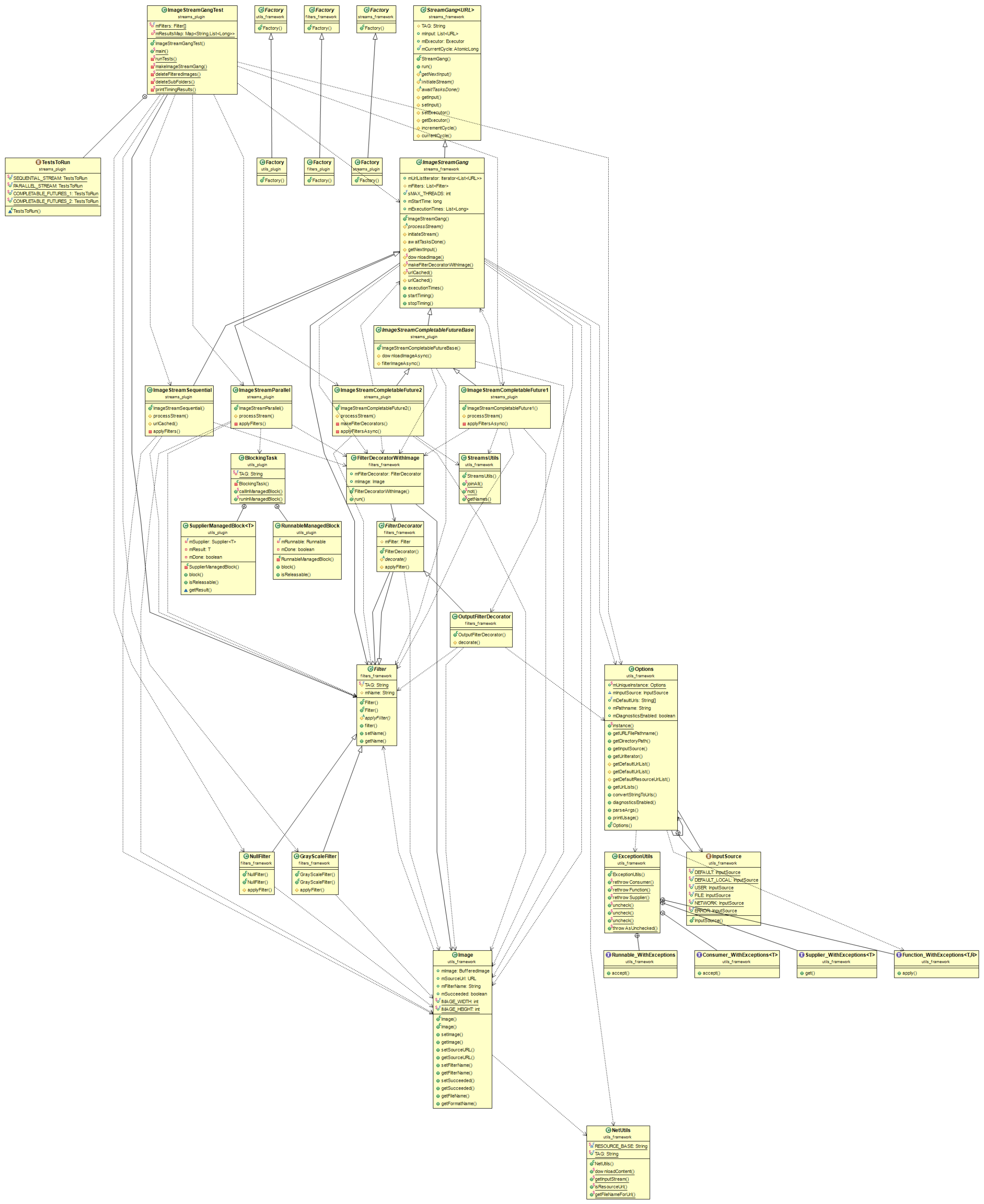


Figure C.8: `ImageStreamGangApp` Java Program. Class diagram after framework and plug-in transformation.

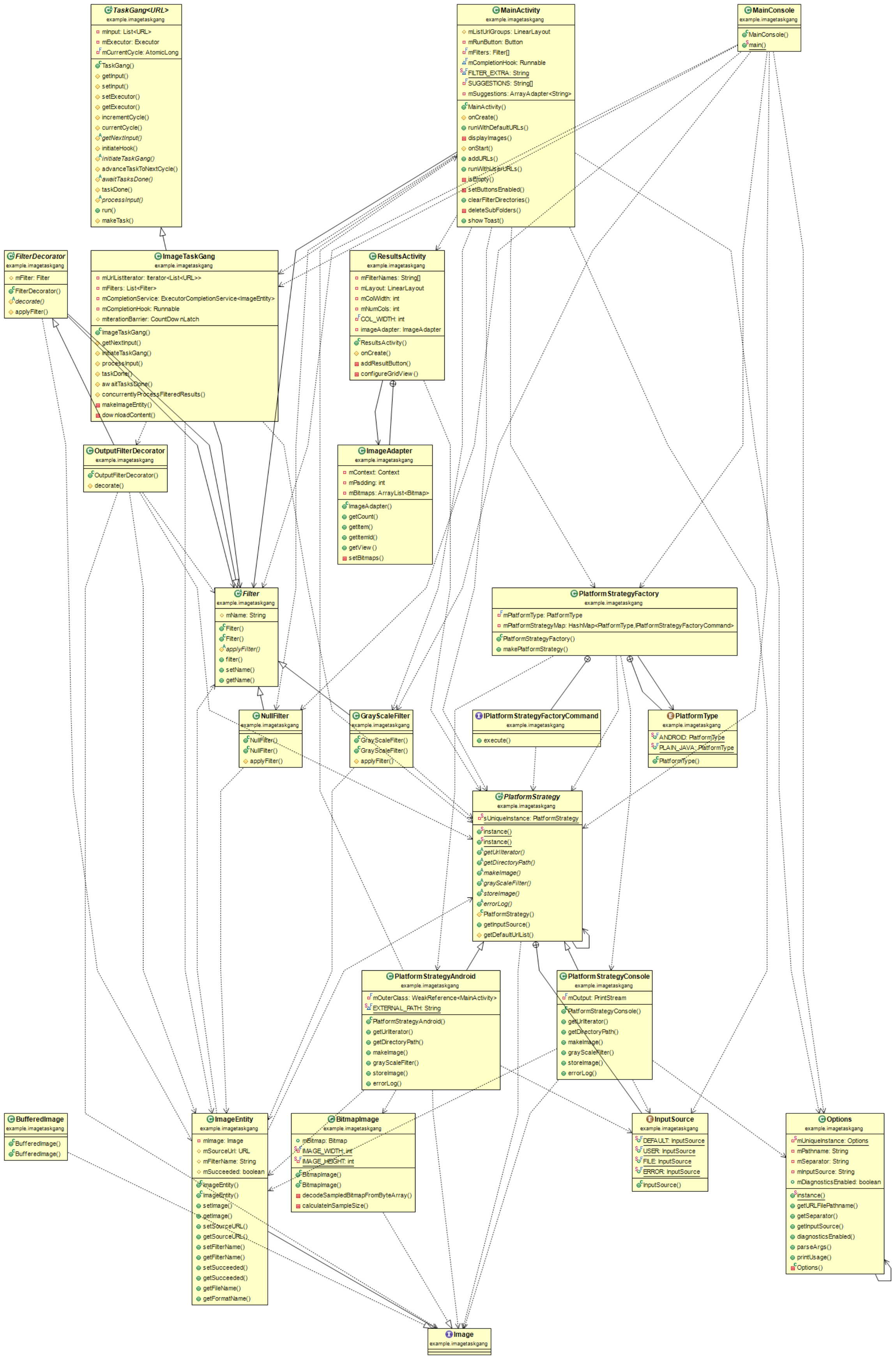


Figure C.9: `ImageTaskGangApplication` Java Program. Original class diagram.

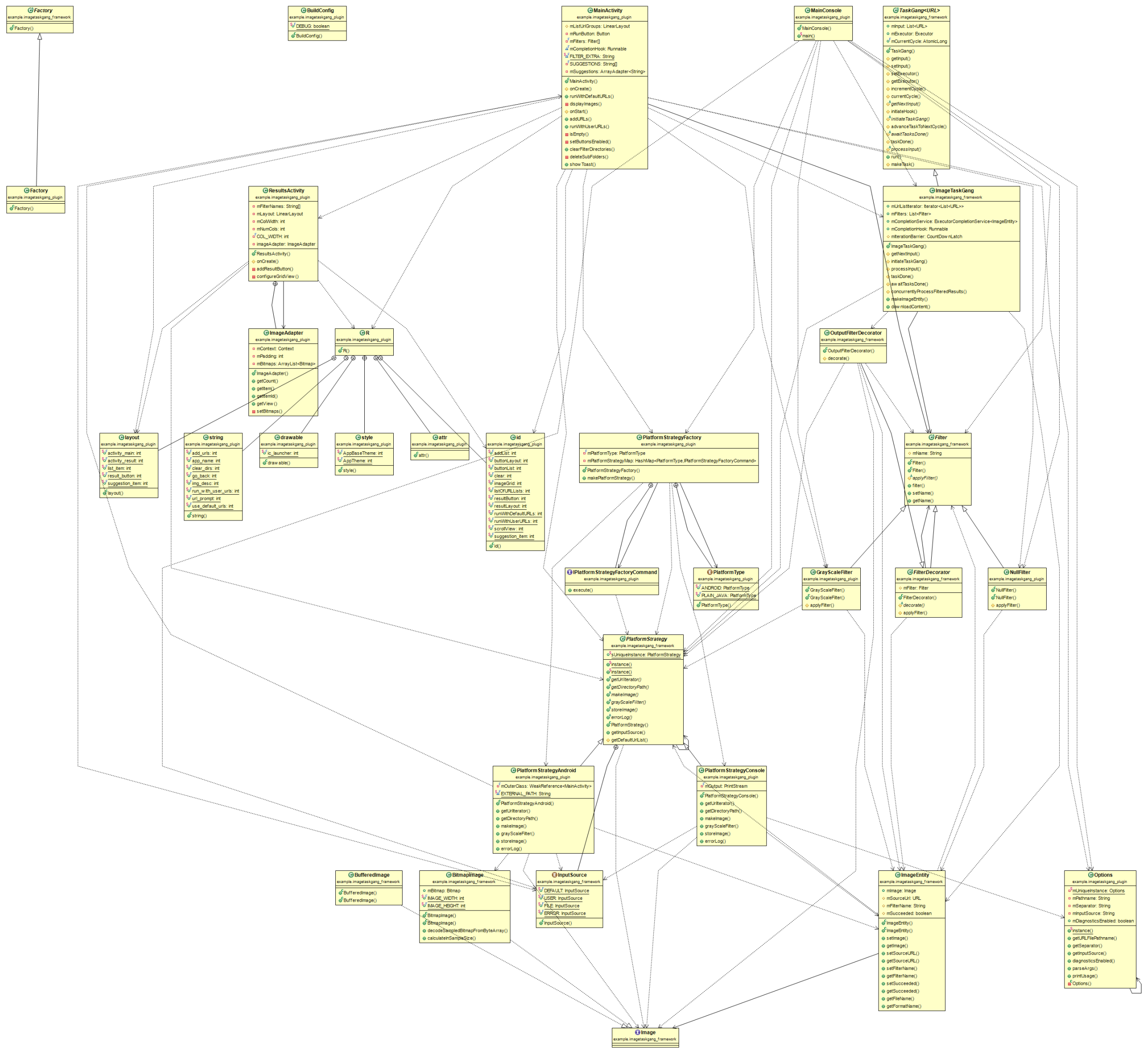


Figure C.10: `ImageTaskGangApplication` Java Program. Class diagram after framework and plug-in transformation.

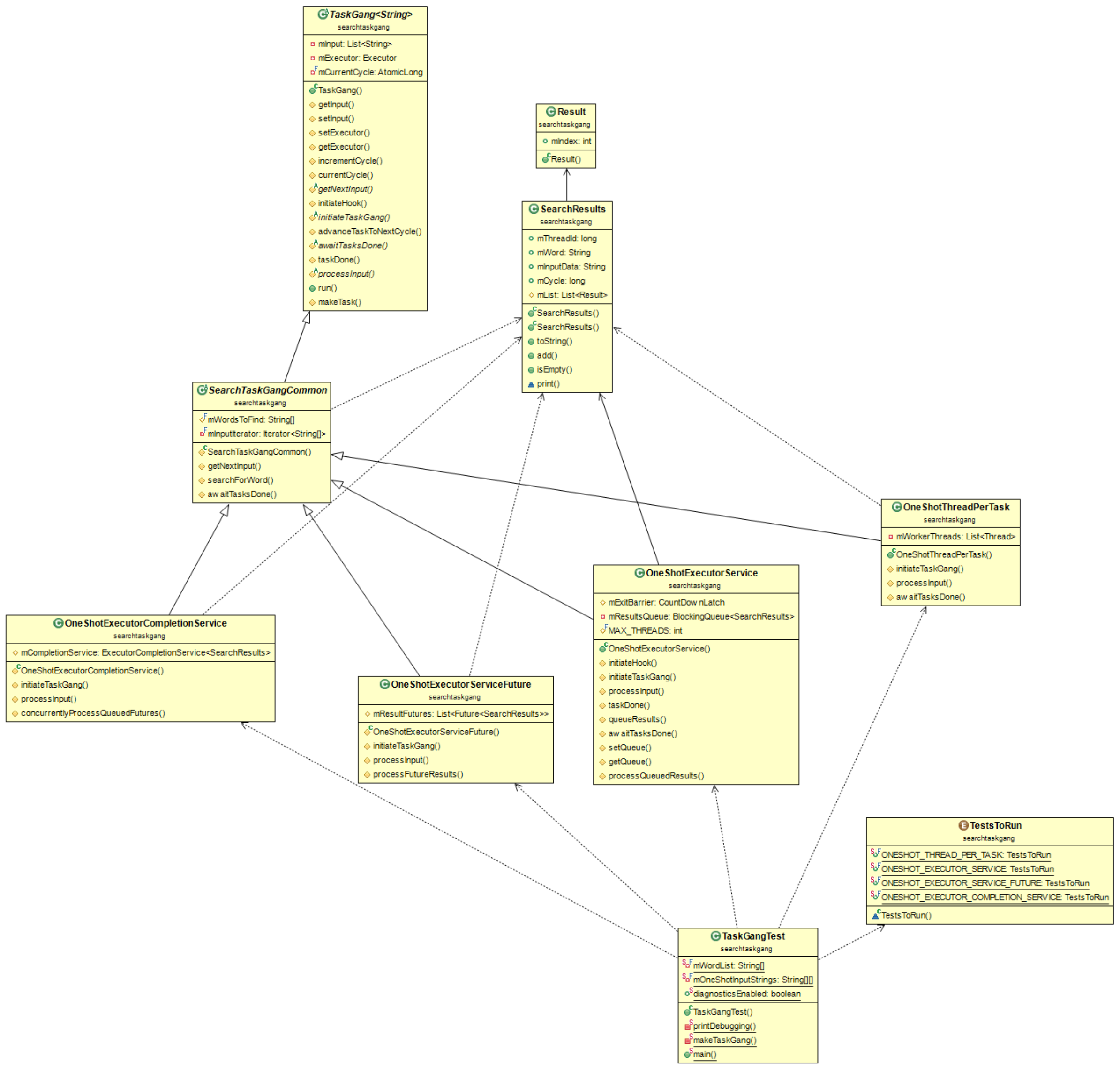


Figure C.11: `SearchTaskGang` Java Program. Original class diagram.

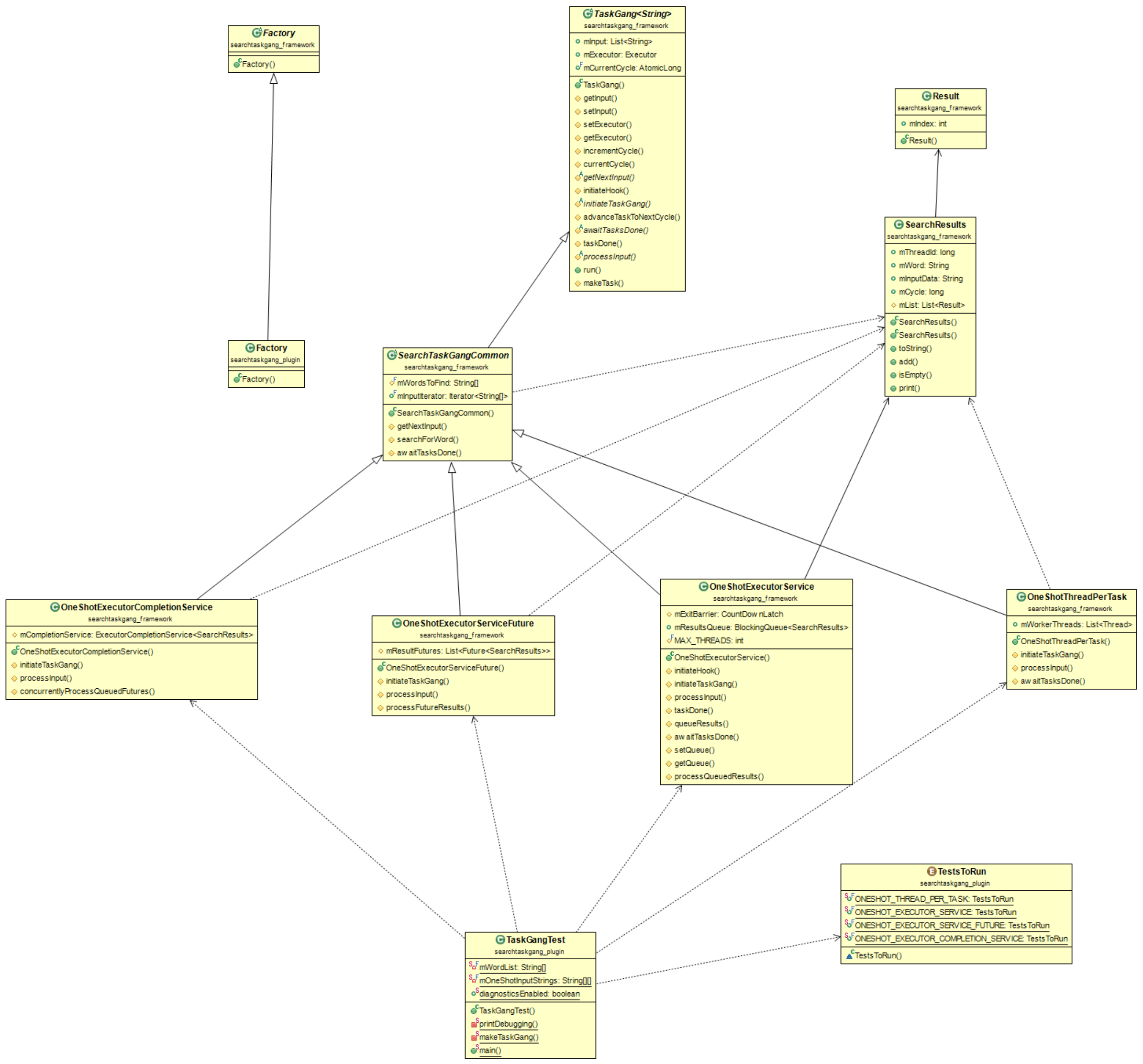


Figure C.12: `SearchTaskGang` Java Program. Class diagram after framework and plug-in transformation.

Appendix D

GAMA platform source code modification

We cloned GAMA platform source code from their open source GitHub repository[1] on August 2016. The class we modified is `GamlModelBuilder`, located in `/msi.gama.lang.gaml/src/msi/gama/lang/gaml/resource/` path. The final source code of this class can be observed in Listing D.1. We made the following changes:

- Located in line 156 is method `getInfo`. This method is executed when a user clicks on `Refresh metadata` option, in a model's contextual menu. We added a call to method `bttf_getCRG` in line 171. `bttf_getCRG` is one of the new methods we implemented in this class.
- From line 211 to the end of the file, a reader can found the methods we implemented to traverse a model's AST, get information about its declarations and their relationships, and finally save them in a CSV file. Figure D.1 shows an example of this CSV output file. It contains all the reference relationships found in a GAML model. This contents serve as input for B$^{t}_{t}$F, which processes them to create a CRG.

These changes were made to our local repository copy, and were not pushed to Gama platform repository.

Listing D.1: `GamlModelBuilder` class source code.

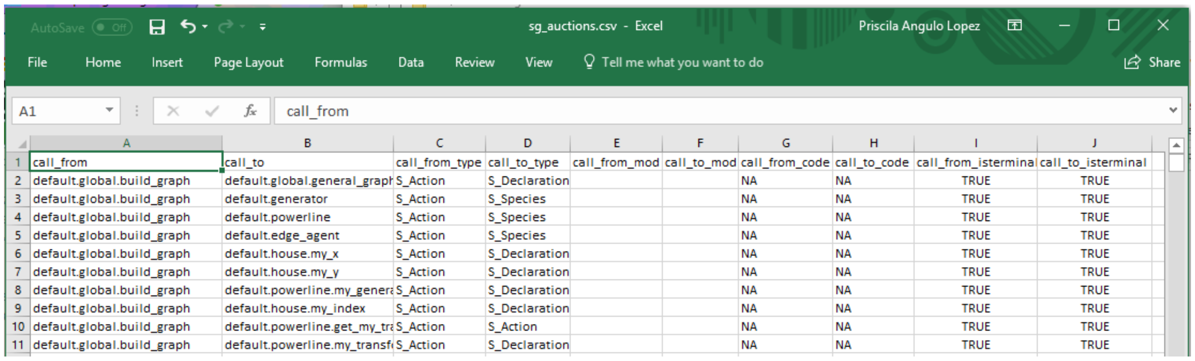
```
1 /*********************************************************************************
2  *
3  *
4  * 'GamlResourceBuilder.java', in plugin 'msi.gama.lang.gaml', is part of the source
       code of the
5  * GAMA modeling and simulation platform.
6  * (c) 2007-2014 UMI 209 UMMISCO IRD/UPMC & Partners
7  *
8  * Visit https://code.google.com/p/gama-platform/ for license information and
       developers contact.
```

[1] `https://github.com/gama-platform/gama/`

```java
 *
 *
 **************************************************************************/
package msi.gama.lang.gaml.resource;

import java.io.*;
import java.util.*;
import org.eclipse.emf.common.util.*;
import org.eclipse.emf.ecore.EAttribute;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.xtext.resource.XtextResourceSet;
import gnu.trove.set.hash.TLinkedHashSet;
import msi.gama.common.interfaces.IKeyword;
import msi.gama.kernel.model.IModel;
import msi.gama.lang.gaml.gaml.*;
import msi.gama.lang.gaml.gaml.impl.ActionDefinitionImpl;
import msi.gama.lang.utils.EGaml;
import msi.gama.util.file.GAMLFile;
import msi.gaml.compilation.*;
import msi.gaml.descriptions.*;

/**
 * Class GamlResourceBuilder.
 *
 * @author drogoul
 * @since 8 avr. 2014
 *
 */
public class GamlModelBuilder implements IModelBuilder {

 XtextResourceSet buildResourceSet = new XtextResourceSet();

 GamlResource fakeResource;
 static URI fakeURI = URI.createURI("temp_builder.gaml", false);

 // private static GamlModelBuilder instance = new GamlModelBuilder();
 //
 // public static GamlModelBuilder getInstance() {
 // return instance;
 // }

 public GamlModelBuilder() {

  buildResourceSet.setClasspathURIContext(GamlModelBuilder.class);
 };

 /**
  * Validates the GAML model inside the resource and returns an ErrorCollector
      (which can later be probed for
  * internal errors, imported errors, warnings or infos)
  * @param resource must not be null
  * @return an instance of ErrorCollector (never null)
  */
```

```
@Override
public ErrorCollector validate(final Resource resource) {
 GamlResource r = (GamlResource) resource;
 r.validate(resource.getResourceSet());
 return r.getErrorCollector();
}

@Override
public ErrorCollector validate(final URI resource) {
 try {
  GamlResource r = (GamlResource) buildResourceSet.createResource(resource);
  return validate(r);
 } finally {
  buildResourceSet.getResources().clear();
 }
}

/**
 * Builds an IModel from the resource.
 * @param resource must not be null
 * @return an instance of IModel or null if the validation has returned errors (use
    validate(GamlResource) to
 * retrieve them if it is the case, or use the alternate form).
 */
@Override
public IModel compile(final Resource resource) {
 return compile(resource.getURI());
}

@Override
public IModel compile(final URI uri) {
 return compile(uri, new ArrayList());
}

/**
 * Builds an IModel from the resource, listing all the errors, warnings and infos
    that occured
 * @param resource must not be null
 * @param a list of errors, warnings and infos that occured during the build. Must
    not be null and must accept the
 * addition of new elements
 * @return an instance of IModel or null if the validation has returned errors.
 */
@Override
public IModel compile(final Resource resource, final List<GamlCompilationError>
    errors) {
 return compile(resource.getURI(), errors);
}

@Override
public IModel compile(final URI uri, final List<GamlCompilationError> errors) {
 try {
  GamlResource r = (GamlResource) buildResourceSet.createResource(uri);
  return r.build(r.getResourceSet(), errors);
 } finally {
  buildResourceSet.getResources().clear();
 }
}

/**
 * Creates a model from an InputStream (which can represent the contents of a file
     or a string. Be aware that all the context will be lost when using this
     method, i.e. paths relative to the model
 * being compiled will be resolved against the a fake URI
 * @see msi.gama.common.interfaces.IModelBuilder#compile(java.io.InputStream,
     java.util.List)
 */

@Override
public IModel compile(final InputStream contents, final List<GamlCompilationError>
    errors) {
 try {
  getFreshResource().load(contents, null);
  return compile(fakeResource, errors);
 } catch (Exception e1) {
  e1.printStackTrace();
  return null;
 }
}

private synchronized GamlResource getFreshResource() {
 if ( fakeResource == null ) {
  fakeResource = (GamlResource) buildResourceSet.createResource(fakeURI);
 } else {
  fakeResource.unload();
 }
 return fakeResource;
}

@Override
public ModelDescription buildModelDescription(final URI uri, final
    List<GamlCompilationError> errors) {
 try {
  GamlResource r = (GamlResource) buildResourceSet.createResource(uri);
  return r.buildDescription(r.getResourceSet(), errors);
 } finally {
  buildResourceSet.getResources().clear();
 }
}

// private static final Set<String> EXTS =
    GamaFileType.extensionsToFullType.keySet();

@Override
public GAMLFile.GamlInfo getInfo(final URI uri, final long stamp) {
 /* Synchronized */XtextResourceSet infoResourceSet = new /* Synchronized
     */XtextResourceSet();
 try {

  GamlResource r = (GamlResource) infoResourceSet.createResource(uri);
  r.load(Collections.EMPTY_MAP);
  TreeIterator<EObject> tree = r.getAllContents();
  Set<String> imports = new TLinkedHashSet();
  Set<String> uses = new TLinkedHashSet();
  Set<String> exps = new TLinkedHashSet();

  /*
   * BttF code
   * This method has been embedded in the "Refresh metadata" option for a GAML
       program
   */
  bttf_getCRG(r);


  while (tree.hasNext()) {
   EObject e = tree.next();
   if ( e instanceof StringLiteral ) {
    String s = ((StringLiteral) e).getOp();
    if ( s.length() > 4 ) {
     URI u = URI.createFileURI(s);
     String ext = u.fileExtension();
     if ( GamaBundleLoader.HANDLED_FILE_EXTENSIONS.contains(ext) ) {
      uses.add(s);
     }
    }
   } else if ( e instanceof S_Experiment ) {
    String s = ((S_Experiment) e).getName();
    Map<String, Facet> f = EGaml.getFacetsMapOf((Statement) e);
    Facet typeFacet = f.get(IKeyword.TYPE);
    if ( typeFacet != null ) {
     String type = EGaml.getKeyOf(typeFacet.getExpr());
     if ( IKeyword.BATCH.equals(type) ) {
      s = GAMLFile.GamlInfo.BATCH_PREFIX + s;
     }
    }
    exps.add(s);
   } else if ( e instanceof Import ) {
    imports.add(((Import) e).getImportURI());
    tree.prune();
   }
  }

  return new GAMLFile.GamlInfo(stamp, imports, uses, exps);
 } catch (IOException e1) {
  e1.printStackTrace();
  return null;
 } finally {
  infoResourceSet.getResources().clear();
 }
}

/*
```

```java
 * BttF related methods implementation starts here...
 */
private void bttf_getCRG(GamlResource codeResource){
 boolean debug = false;

 String program_name =
     bttf_getModelName(codeResource.getAbsolutePath().toOSString());
 ArrayList<BttF_Declaration> dec_list = new ArrayList<BttF_Declaration>();
 ArrayList<Bttf_Reference> ref_list = new ArrayList<Bttf_Reference>();

 bttf_getMainContainmentTree(
   codeResource.getAllContents(),
   "",
   dec_list,
   ref_list);

 bttf_getReferencesFile(program_name, dec_list, ref_list);

 if(debug){
  System.out.println("***********************************************");
  System.out.println(program_name);
  System.out.println("Start getting tree");
  System.out.println(dec_list.toString());
  System.out.println(ref_list.toString());
  System.out.println("End getting tree");
 }
}

private String bttf_getModelName(String abs_path){
 int idxSlash = abs_path.lastIndexOf('\\');
 int idxDot = abs_path.lastIndexOf('.');
 if (idxSlash != -1 && idxDot != -1 && idxSlash < idxDot){
  return abs_path.substring(idxSlash+1, idxDot);
 }
 return "";
}

private void bttf_getMainContainmentTree(TreeIterator<EObject> tree, String parent,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 while (tree.hasNext()) {
  EObject e = tree.next();
  if ( e instanceof S_Global ){
   bttf_getElement((S_Global)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Species ){
   bttf_getElement((S_Species)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Experiment ) {
   bttf_getElement((S_Experiment)e, parent, dec_list, ref_list);
  } if ( e instanceof Import ) {
   bttf_getElement((Import)e, parent, dec_list, ref_list);
  }
 }
}

private void bttf_getContainmentTree(TreeIterator<EObject> tree, String parent,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 while (tree.hasNext()) {
  EObject e = tree.next();
  if ( e instanceof S_Global ){
   bttf_getElement((S_Global)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Species ){
   bttf_getElement((S_Species)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Reflex ) {
   bttf_getElement((S_Reflex)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Action ) {
   bttf_getElement((S_Action)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Var ) {
   bttf_getElement((S_Var)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Declaration ) {
   bttf_getElement((S_Declaration)e, parent, dec_list, ref_list);
  } if ( e instanceof S_Experiment ) {
   bttf_getElement((S_Experiment)e, parent, dec_list, ref_list);
  } if ( e instanceof Import ) {
   bttf_getElement((Import)e, parent, dec_list, ref_list);
  }
 }
}

private void bttf_getReferenceTree(TreeIterator<EObject> tree, String fromWCont,
    ArrayList<Bttf_Reference> ref_list){
 while (tree.hasNext()) {
  EObject e = tree.next();
  if ( e instanceof VariableRef ) {
   bttf_getReference((VariableRef)e, fromWCont, ref_list);
  } if ( e instanceof SkillRef ) {
   bttf_getReference((SkillRef)e, fromWCont, ref_list);
  } if ( e instanceof ActionRef ) {
   bttf_getReference((ActionRef)e, fromWCont, ref_list);
  }
 }
}


private void bttf_getElement(S_Global e, String parent, ArrayList<BttF_Declaration>
    dec_list, ArrayList<Bttf_Reference> ref_list){
 bttf_getContainmentTree(e.eAllContents(), parent + "global" + ".", dec_list,
     ref_list);
}

private void bttf_getElement(S_Species e, String parent,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Species," + parent + "." +e.getName() + "," + codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Species", parent + e.getName(),
      codeName);
  dec_list.add(dec);

  bttf_getContainmentTree(e.eAllContents(), parent + e.getName() + ".", dec_list,
      ref_list);
  bttf_getReferenceTree(e.eAllContents(), codeName, ref_list);

 }
}

private void bttf_getElement(S_Reflex e, String parent, ArrayList<BttF_Declaration>
    dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Reflex," + parent + "."+ e.getName() + "," + codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Reflex", parent + e.getName(),
      codeName);
  dec_list.add(dec);

  bttf_getReferenceTree(e.eAllContents(), codeName, ref_list);
 }
}

private void bttf_getElement(S_Action e, String parent, ArrayList<BttF_Declaration>
    dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Action," + parent + "." + e.getName() + "," + codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Action", parent + e.getName(),
      codeName);
  dec_list.add(dec);

  bttf_getReferenceTree(e.eAllContents(), codeName, ref_list);
 }
}

private void bttf_getElement(S_Var e, String parent, ArrayList<BttF_Declaration>
    dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Var," + parent + "." + e.getName() + "," + codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Var", parent + e.getName(),
      codeName);
  dec_list.add(dec);

 }
}

private void bttf_getElement(S_Declaration e, String parent,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null
   &&
       e.getClass().getName().equals("msi.gama.lang.gaml.gaml.impl.S_DefinitionImpl")){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Declaration," + parent + "." + e.getName() + "," +
      codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Declaration", parent +
      e.getName(), codeName);
  dec_list.add(dec);
 }
}


private void bttf_getElement(S_Experiment e, String parent,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("S_Experiment," + parent + "." + e.getName() + "," +
      codeName);
  BttF_Declaration dec = new BttF_Declaration("S_Experiment", parent + e.getName(),
      codeName);
  dec_list.add(dec);

  bttf_getContainmentTree(e.eAllContents(), parent + e.getName() + ".", dec_list,
      ref_list);
  bttf_getReferenceTree(e.eAllContents(), codeName, ref_list);
 }
}

private void bttf_getElement(Import e, String parent, ArrayList<BttF_Declaration>
    dec_list, ArrayList<Bttf_Reference> ref_list){
 if(e != null && e.getName() != null){
  String codeName = e.eContainer().hashCode() + "." + e.getName();

  //System.out.println("Import," + parent + "." + e.getName() + "," + codeName);
  BttF_Declaration dec = new BttF_Declaration("Import", parent + e.getName(),
      codeName);
  dec_list.add(dec);
 }
}

private void bttf_getReference(VariableRef e, String fromWCont,
    ArrayList<Bttf_Reference> ref_list){
 if(fromWCont != null && e != null && e.getRef()!= null){
   if (e.getRef() instanceof S_Species){
    bttf_getReference("VariableRef-S_Species", fromWCont, e.getRef(), ref_list);
   }
   else if (e.getRef() instanceof S_Reflex){
    bttf_getReference("VariableRef-S_Reflex", fromWCont, e.getRef(), ref_list);
   }
   else if (e.getRef() instanceof S_Action){
    bttf_getReference("VariableRef-S_Action", fromWCont, e.getRef(), ref_list);
   }
   else if (e.getRef() instanceof S_Var){
    bttf_getReference("VariableRef-S_Var", fromWCont, e.getRef(), ref_list);
   }
   else if (e.getRef() instanceof S_Declaration){
    if(!(e.getRef().getClass().getName()
                 .equals("msi.gama.lang.gaml.gaml.impl.S_LoopImpl"))){
     bttf_getReference("VariableRef-S_Declaration", fromWCont, e.getRef(),
         ref_list);
    }
   }
 }
}

private void bttf_getReference(String refDesc, String fromWCont, VarDefinition r,
    ArrayList<Bttf_Reference> ref_list){
 if(fromWCont != null && r != null && r.getName() != null){

  /*System.out.println(refDesc
    + "," + from + " -> " + r.getName()
    + "," + fromWCont + " -> " + r.eContainer().hashCode() + "." + r.getName());*/

  Bttf_Reference ref = new Bttf_Reference(fromWCont, r.eContainer().hashCode() +
      "." + r.getName());
  if(!ref_list.contains(ref)){
   ref_list.add(ref);
  }
 }
}

private void bttf_getReference(SkillRef e, String fromWCont,
    ArrayList<Bttf_Reference> ref_list){
 if(fromWCont != null && e != null && e.getRef()!= null){
  /*System.out.println("SkillRef"
    + "," + from + " -> "+ e.getRef().getName()
    + "," + fromWCont + " -> " + e.getRef().eContainer().hashCode() + "." +
        e.getRef().getName());*/

  Bttf_Reference ref = new Bttf_Reference(fromWCont,
      e.getRef().eContainer().hashCode() + "." + e.getRef().getName());
  if(!ref_list.contains(ref)){
   ref_list.add(ref);
  }
 }
}

private void bttf_getReference(ActionRef e, String fromWCont,
    ArrayList<Bttf_Reference> ref_list){
 if(fromWCont != null && e != null && e.getRef()!= null &&
   !(e.getRef().getClass().getName()
              .equals("msi.gama.lang.gaml.gaml.impl.ActionDefinitionImpl")) ){

  /*System.out.println("ActionRef"
   + "," + from + " -> " + e.getRef().getName()
   + "," + fromWCont + " -> " + e.getRef().eContainer().hashCode() + "." +
       e.getRef().getName());*/

  Bttf_Reference ref = new Bttf_Reference(fromWCont,
      e.getRef().eContainer().hashCode() + "." + e.getRef().getName());
  if(!ref_list.contains(ref)){
   ref_list.add(ref);
  }
 }
}

class Bttf_Reference{
 String from;
 String to;

 public Bttf_Reference(String from, String to) {
  this.from = from;
  this.to = to;
 }

 @Override
 public String toString() {
  return from + " -> " + to + "\n";
 }

 @Override
    public boolean equals(Object object)
    {
        if (object != null && object instanceof Bttf_Reference)
        {
            if( this.from.equals( ((Bttf_Reference)object).from) && this.to.equals(
                ((Bttf_Reference)object).to) ){
             return true;
            }
        }

        return false;
    }



}

class BttF_Declaration{
 String type;
 String name;
 String code_name;

 public BttF_Declaration(String type, String name, String code_name) {
  this.type = type;
  this.name = name;
  this.code_name = code_name;
 }

 @Override
 public String toString() {
  return "BttF_Declaration [type=" + type + ", name=" + name + ", code_name=" +
      code_name + "]\n";
 }


}


private void bttf_getReferencesFile(String program_name,
    ArrayList<BttF_Declaration> dec_list, ArrayList<Bttf_Reference> ref_list){
 String file_path = System.getProperty("user.home") + "/Desktop/" + program_name +
     ".csv";
 String header =
     "call_from,call_to,call_from_type,call_to_type,call_from_mod,call_to_mod,"
   + "call_from_code,call_to_code,call_from_isterminal,call_to_isterminal\r\n";

 BufferedWriter writer = null;

 try{
  //write feature model - partition task
  writer = new BufferedWriter(new FileWriter(file_path, false));
  writer.append(header);
  for(Bttf_Reference r : ref_list){
   BttF_Declaration from = bttf_findDeclaration(r.from, dec_list);
   BttF_Declaration to = bttf_findDeclaration(r.to, dec_list);
   writer.append(getCsvReferenceLine(from, to));
  }
  writer.flush();
  System.out.println("CRG csv file saved on: " + file_path);
 }catch (IOException e){
  e.printStackTrace();
 }
 finally{
     try{
         if ( writer != null)
         writer.close( );
     }catch ( IOException e){
      e.printStackTrace();
     }
 }
}

private BttF_Declaration bttf_findDeclaration(String code_name,
    ArrayList<BttF_Declaration> dec_list){
 for(BttF_Declaration d : dec_list){
  if(d.code_name.equals(code_name)){
   return d;
  }
 }
 return null;
}

/*
 * required fields
 * gotten from bttf.Reference class
 * private String call_from;
 private String call_to;
 private ElementType call_from_type;
 private ElementType call_to_type;
 private String call_from_mod;
 private String call_to_mod;
 private String call_from_code;
 private String call_to_code;
 private boolean call_from_isterminal;
 private boolean call_to_isterminal;
 */
private String getCsvReferenceLine(BttF_Declaration from, BttF_Declaration to){
 if(from != null && to != null){
  StringBuffer sb = new StringBuffer();
  sb.append("default." + from.name + ","); //call_from
  sb.append("default." + to.name + ","); //call_to
  sb.append(from.type + ","); //call_from_type
  sb.append(to.type + ","); //call_to_type
  sb.append(","); //call_from_mod
  sb.append(","); //call_to_mod
  sb.append("NA,"); //call_from_code
  sb.append("NA,"); //call_to_code
  sb.append("TRUE,"); //call_from_isterminal
  sb.append("TRUE\r\n"); //call_to_isterminal
  return sb.toString();
 }
 return "";
}

/*
 * BttF related NOTES:
 * TypeRef is a reference to a primitive type
 * S_Entities instances weren't found
 * S_Environment instances weren't found
 * Function instanced didn't make sense
 */
}
```



	A	B	C	D	E	F	G	H	I	J
1	call_from	call_to	call_from_type	call_to_type	call_from_mod	call_to_mod	call_from_code	call_to_code	call_from_istermina	call_to_isterminal
2	default.global.build_graph	default.global.general_graph	S_Action	S_Declaration			NA	NA	TRUE	TRUE
3	default.global.build_graph	default.generator	S_Action	S_Species			NA	NA	TRUE	TRUE
4	default.global.build_graph	default.powerline	S_Action	S_Species			NA	NA	TRUE	TRUE
5	default.global.build_graph	default.edge_agent	S_Action	S_Species			NA	NA	TRUE	TRUE
6	default.global.build_graph	default.house.my_x	S_Action	S_Declaration			NA	NA	TRUE	TRUE
7	default.global.build_graph	default.house.my_y	S_Action	S_Declaration			NA	NA	TRUE	TRUE
8	default.global.build_graph	default.powerline.my_genera	S_Action	S_Declaration			NA	NA	TRUE	TRUE
9	default.global.build_graph	default.house.my_index	S_Action	S_Declaration			NA	NA	TRUE	TRUE
10	default.global.build_graph	default.powerline.get_my_tra	S_Action	S_Action			NA	NA	TRUE	TRUE
11	default.global.build_graph	default.powerline.my_transf	S_Action	S_Declaration			NA	NA	TRUE	TRUE

Figure D.1: CSV output example for a GAML model.

Bibliography

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines.* Springer, 2013.

[2] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[3] Sven Apel, Sergiy Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, 2012.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, September 2005.

[5] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *ICSR*, 1998.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng.*, June 2004.

[7] Don Batory. Software design course. Last visited on 10/11/2017.

[8] Don Batory. Private conversation with Marouane Kessentini, 2017.

[9] Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In *Software Product Lines*, 2000.

[10] Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In *Software Product Line Conference*, pages 227–247. Springer, 2000.

[11] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Trans. Softw. Eng. Methodol.*, April 2002.

[12] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, October 1992.

[13] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *ICSE*, 2004.

[14] Allan G Bluman. *Elementary statistics*. McGraw Hill Publishers, 2000.

[15] Jan Bosch. Object-oriented frameworks: Problems & experiences, 1997.

[16] Ramon F Brena, Carl W Handlin, and Priscila Angulo. A smart grid electricity market with multiagents, smart appliances and combinatorial auctions. In *Smart Cities Conference (ISC2), 2015 IEEE First International*, pages 1–6. IEEE, 2015.

[17] Catalog of software product lines. `http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm`.

[18] Luca Cernuzzi, Thomas Juan, Leon Sterling, and Franco Zambonelli. The gaia methodology. In *Methodologies and Software Engineering for Agent Systems*, pages 69–88. Springer, 2004.

[19] Christina Chavez. *A model-driven approach for aspect-oriented design*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2004.

[20] D Manning Christopher, Raghavan Prabhakar, and SCHÜTZE Hinrich. Introduction to information retrieval. *An Introduction To Information Retrieval*, 151:177, 2008.

[21] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific frameworks. *CACM*, October 1997.

[22] Massimo Cossentino and Colin Potts. Passi: A process for specifying and implementing multi-agent systems using uml. *Citeseer 2002*, 2002.

[23] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, 2011.

[24] John Dalbey. Java lines of code counter. Last visited on 10/11/2017.

[25] J Dehlinger and RR Lutz. A product-line requirements approach to safe reuse in multi-agent systems. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[26] J Dehlinger and RR Lutz. Gaia-PL: A product line engineering approach for efficiently designing multiagent systems. *ACM Transactions on Software Engineering and Methodology*, 20(4):Article 17, 2011.

[27] Jay L Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage learning, 2011.

[28] Edsger W. Dijkstra. The structure of the &ldquo;the&rdquo;-multiprogramming system. In *SOSP*, 1967.

[29] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 2013.

[30] Alexis Drogoul, Edouard Amouroux, Philippe Caillou, Benoit Gaudou, Arnaud Grignard, Nicolas Marilleau, Patrick Taillandier, Maroussia Vavasseur, Duc-An Vo, and Jean-Daniel Zucker. Gama: multi-level and complex environment for agent-based models and simulations. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1361–1362. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[31] Eclipse. Abstract syntax tree.

[32] Eclipse. Refactor actions Java development user guide. Last visited on 10/10/2017.

[33] Eclipse Java development tools (JDT). `eclipse.org/jdt/`.

[34] Stephan Erb. A survey of software refactoring tools. *Master's thesis, Baden-Wrttemberg Cooperative State University, Karlsruhe*, 2010.

[35] Martin Faunes, Marouane Kessentini, and Houari Sahraoui. Deriving high-level abstractions from legacy software using example-driven clustering. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 188–199. IBM Corp., 2011.

[36] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *CACM*, October 1997.

[37] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[38] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[39] R. Fuhrer, Markus Keller, and A. Kiezun. Advanced refactoring in the eclipse jdt: Past, present, and future. In *ECOOP Workshop on Refactoring Tools (WRT)*, 2007.

[40] NR Genza and ES Mighele. Review on multi-agent oriented software engineering implementation. *International Journal of Computer and Information Technology*, 2(03):511–520, 2013.

[41] H Gomaa and ME Shin. Variability Modeling in Model-Driven Software Product Line Engineering. In *2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010)*, pages 25–36, Paris, France, 2010.

[42] Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, and Alexis Drogoul. Gama 1.6: Advancing the art of complex agent-based modeling and simulation. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 117–131. Springer, 2013.

[43] A. N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *CACM*, May 1976.

[44] Nicholas R. Jennings and Michael Wooldridge. Agent-Oriented Software Engineering. *ARTIFICIAL INTELLIGENCE*, 117:277–296, 2000.

[45] Ralph Johnson and Bryan Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[46] Ralph E Johnson. Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[47] Christian Kastner, Alexander Dreiling, and Klaus Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE TSE*, January 2014.

[48] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014.

[49] Jongwook Kim, Don Batory, and Danny Dig. Scripting parametric refactorings in java to retrofit design patterns. In *ICSME*, 2015.

[50] Jongwook Kim, Don Batory, and Danny Dig. Refactoring java software product lines. SPLC, 2017.

[51] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1145–1156. ACM, 2016.

[52] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[53] Uirá Kulesza, Alessandro Garcia, and Carlos Lucena. An aspect-oriented generative approach. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 166–167. ACM, 2004.

[54] Uirá Kulesza, Alessandro Garcia, Carlos Lucena, and Paulo Alencar. A generative approach for multi-agent system development. In *International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 52–69. Springer, 2004.

[55] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware Refactoring in the Wild. In *ICSE*, 2015.

[56] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. *Software & Systems Modeling*, 2016.

[57] Marcus Eduardo Markiewicz and Carlos JP de Lucena. Object oriented framework development. *Crossroads*, 7(4):3–9, 2001.

[58] Wolfram MathWorld. Correlation coefficient, 2017.

[59] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009.

[60] G. S. Mudholkar. Fisher's z-transformation. In *Encyclopedia of Statistical Sciences*. John Wiley and Sons, 2006.

[61] Jörg P Müller and Klaus Fischer. Application impact of multi-agent systems and technologies: A survey. In *Agent-oriented software engineering*, pages 27–53. Springer, 2014.

[62] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5), 2008.

[63] Sarah Nadi and et al. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

[64] NetBeans 8.0.2. `https://netbeans.org/`.

[65] I Nunes, U Kulesza, and C Nunes. Extending PASSI to Model Multi-agent Systems Product Lines. In *SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing*, pages 729–730, 2009.

[66] I Nunes, U Kulesza, C Nunes, and CJP Lucena. A domain engineering process for developing multi-agent systems product lines. In *AAMAS '09 Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 1339–1340, Budapest, Hungary, 2009.

[67] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, U. of Illinois Urbana-Champaign, 1992.

[68] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[69] Oracle. Enum types. The Java Tutorials. Last visited on 09/26/2017.

[70] Oracle. Lesson: Annotations. The Java Tutorials. Last visited on 09/26/2017.

[71] Oracle. Understanding class members. The Java Tutorials. Last visited on 09/26/2017.

[72] Leonardo Teixeira Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. Feature scattering in the large: a longitudinal study of linux kernel device drivers. In *Modularity*, 2015.

[73] J Pena, MG Hinchey, and M Resinas. Managing the Evolution of an Enterprise Architecture Using a MAS-Product-Line Approach. In *IWSSAâĂŹ06 5th International Workshop on System/Software Architectures*, 2006.

[74] J Pena, MG Hinchey, M Resinas, Roy Sterrittc, and James L. Rashb. Designing and managing evolving systems using a MAS product line approach. *Science of Computer Programming*, 66(1):71–86, 2007.

[75] J Pena, MG Hinchey, and A Ruiz-Cortés. Multi-agent system product lines: challenges and benefits. *Communications of the ACM*, 49(12):82–85, 2006.

[76] Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, and Wenyun Zhao. Improving feature location using structural similarity and iterative graph mapping. *Journal of Systems and Software*, 86(3):664–676, 2013.

[77] GAMA Platform. Organization of a model. Last visited on 10/24/2017.

[78] Klaus Pohl, Günter Böckle, and Frank Van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, Berlin/Heidelberg, 2005.

[79] Wolfgang Pree. *Design Patterns for Object-oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., 1995.

[80] Eclipse project. AspectJ. Last visited on 10/28/2017.

[81] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[82] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[83] Nornadiah Mohd Razali, Yap Bee Wah, et al. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1):21–33, 2011.

[84] William Revelle. *psych: Procedures for Psychological, Psychometric, and Personality Research*. Northwestern University, Evanston, Illinois, 2016. R package version 1.6.9.

[85] Martin P Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):18, 2008.

[86] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, 2013.

[87] Andrew Rubinger and Bill Burke. *Enterprise JavaBeans 3.1, 6th Edition*. O'Reilly Media, 2010.

[88] Ruby on Rails. `http://rubyonrails.org/`.

[89] David Saff, Kevin Cooney, Stefan Birkner, and Marc Philipp. JUnit test tool, http://www.junit.org. Last visited on 10/17/2017.

[90] Han Albrecht Schmid. Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51, 1997.

[91] Douglas C. Schmidt. Java source code. LiveLessons: Design Patterns in Java. Last visited on 10/11/2017.

[92] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 2002.

[93] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, April 2002.

[94] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *ECOOP*, 1998.

[95] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.

[96] Thomas Thüm and et al. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, January 2014.

[97] Kai Ming Ting. *Precision and Recall*, pages 781–781. Springer US, Boston, MA, 2010.

[98] M. T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *IEEE TSE*, July 2012.

[99] Marco Tulio Valente, Virgilio Borges, and Leonardo Passos. A semi-automatic approach for extracting software product lines. *IEEE transactions on Software Engineering*, 38(4):737–754, 2012.

[100] Jose M. Vidal. *Fundamentals of Multiagent Systems*. Unpublished, 2006.

[101] Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs, Second Edition*. Addison-Wesley, 2004.

[102] Taiyun Wei and Viliam Simko. *corrplot: Visualization of a Correlation Matrix*, 2016. R package version 0.77.

[103] Eric W. Weisstein. Fisher's z-transformation. From MathWorld—A Wolfram Web Resource.

[104] Wikipedia. Aspect-oriented programming. Last visited on 10/28/2017.

[105] Wikipedia. Dense graph. Last visited on 10/27/2017.

[106] Wikipedia. Eclipse (software).

[107] Wikipedia. Stratego/xt.

[108] WikiPedia Software Design Patterns. `https://en.wikipedia.org/wiki/Software_design_pattern`.

[109] M Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2008.

[110] Xtext. Integration with emf. Last visited on 10/25/2017.

[111] Franco Zambonelli, Nicholas R Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.

Published Papers

Priscila Angulo-Lopez and Guillermo Jimenez-Perez. Collaborative Agents Framework for the Internet of Things. In *Workshop Proceedings of the 8th International Conference on Intelligent Environments*, vol. 13, p. 191. IOS Press, 2012.

Ramon F. Brena, Carl W. Handlin, and Priscila Angulo. A Smart Grid Electricity Market with Multiagents, Smart appliances and Combinatorial Auctions. In *Smart Cities Conference (ISC2)*, 2015 IEEE First International, pp. 1-6. IEEE, 2015.

Priscila Angulo, Claudia Cristina Guzman, Guillermo Jimenez, and David Romero. A Service Oriented Architecture and its ICT-Infrastructure to support Eco-Efficiency Performance Monitoring in Manufacturing Enterprises. *International Journal of Computer Integrated Manufacturing*, vol. 30, no. 1, p. 202. Taylor & Francis, 2016.

Curriculum Vitae

Priscila Angulo López was born in Mazatlán, México, on February 1, 1987. She earned the Computer Systems bachelor's degree from the Universidad de Occidente, Mazatlán Campus in November 2008. After graduating, she worked for three years in a software development company in Monterrey, México.

She was accepted in the graduate program in Information Technologies and Communications in December 2011. She is currently working at Microsoft for Windows division.

Her research interests include graph theory, intelligent systems and high level reuse and customization through feature-oriented programming and software product lines.