First-class Features

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Sagar Sunkle
geb. am 27. September 1980 in Mumbai, Indien

Gutachter:
Prof. Dr. Gunter Saake
Prof. Dr. Uwe Aßmann
Prof. Dr. Olaf Spinczyk

Promotionskolloquium: Magdeburg, den 08. April 2011

Sunkle, Sagar:
First-class Features
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2011.

Abstract

The term *software engineering* originated with the intent to apply engineering principles to the development of software. In the four decades after its birth, a multitude of efforts have been made to counter what has been known as the *software crisis*. Software possesses *essential difficulties* such as complexity, conformity, changeability, and invisibility. To this date, creating good software all the while addressing all of its essential difficulties in a disciplined manner requires substantial effort.

In general, attempts to thwart failures in software development are guided by a set of fundamental principles. The principle of *separation of concerns* is one such principle. The basic idea behind this principle is that any software created to solve a problem needs to address specific issues, called *concerns*, which should be separated and treated in isolation. Such separated concerns are easy to customize, maintain, reuse, and comprehend. This principle led to the development of concepts such as program families and in turn to the concept of *software product lines* (SPLs). Instead of creating a single software system, the stress is given on planning for and enabling an SPL of a software system from which individual software products can be obtained that are suited to specific functionality. Feature-oriented software development (FOSD) is software development paradigm used to create SPLs and focuses on *features* as the mechanism of separation of concerns.

Features represent a specific functionality targeted at satisfying a particular requirement. The process of creating SPLs using FOSD is guided by two principal activities aimed at *modeling* the features of a software system and *implementing* the features thus modeled. In spite of being a relatively young discipline, FOSD is marked by a bewildering array of proposals for modeling and implementing features. The conceptual simplicity of features has led to adopting ideas from every conceivable programming paradigm, language, and tool that can aid in modeling and implementing features. There is a need to reconcile this enormous effort and this would require deciphering the precise roles played by feature modeling and feature implementation in FOSD. This dissertation aspires to contribute in this direction.

While a multitude of feature modeling and feature implementation techniques provide paradigm-level, language-level, and tool-level means to deal with fea-

tures, they do so in strikingly different ways. In this dissertation, we go back to the basics and investigate the nature of features in terms of their representation and composition. We review a variety of ways in which features and related entities are represented and composed to obtain software products. We strive to provide answers to such questions as what kind of representation of features is appropriate from both their modeling and implementation perspective? Is it possible to reconcile representation of features to so that they can be addressed uniformly during modeling and implementation? How to implement a technique so that features from modeling and implementation perspective are represented and composed in a coherent manner?

We attempt to answer these questions in the form of *first-class* features aimed at uniform representation and coherent composition of features from both modeling and implementation perspectives. We implement first-class features in two diverse programming languages. A set of case studies demonstrate the practical applicability of first-class features and also provide further insights into the nature of features in general and first-class features in particular. With the implementations of first-class features and case-studies in them, we show that first-class features aid in a clean separation of feature concern, preserve the conceptual integrity of features due to uniform representation, provide clear traceability links between feature models and feature implementation and support easy accommodation of other extensions of the host language.

Finally, after studying the nature of features closely, we suggest further avenues of development in first-class features and note what we think to be the core issues in FOSD that would attract major research in future.

Acknowledgements

What brought me from India to Germany was my desire to pursue Ph.D. in Computer Science. It has been quite some time since this journey began and it has been a fascinating journey all the way. In this journey, I met many people who traversed the path with me and those who had already reached the destination and thus knew more about it than me and helped me reach it. I want to thank all my companions in this journey now that my final destination is in sight.

First of all, I want to thank my advisor Prof. Gunter Saake. We got introduced during my Master's degree in Data and Knowledge Engineering at the Otto-von-Guericke University of Magdeburg. I used to be flabbergasted by his grasp of concepts and how quickly he could arrive at the most essential part of any subject matter that he was discussing. In time, he called me up to join his group and ever since he has been supportive in most important aspects of my life from then on. He always believed in my abilities and stood by me in all situations.

Next, I want to thank my colleagues Marko Rosenmüller, Norbert Siegmund, Martin Kuhlemann, and Christian Kästner. It was because of them that I got introduced to the field of software product lines in the first place. During my Master's thesis, they helped in strengthening my conceptual understanding of features. Later in the Ph.D., we attended many events together and those were the most enjoyable times filled with countless discussions and tons of fun.

I would also like to thank Sven Apel. We only had a handful of discussions in person during my research. But his enormous body of work always inspired me and worked like a guidepost to show me the path and to encourage me to carry on even when I could not see afar. His influence on my thinking has been capital in the way I approached my the concept of features.

I also want to thank Mario Pukall, Syed Saif ur Rehman, Ateeq Khan, and Ingolf Geist. I had a very fruitful collaboration with Mario in a combined research theme. Saif and Ateeq have always been there, particularly Ateeq who also arrived in Germany at the same time that I did and we have been friends since then. Ingolf has been supportive in a unique way. He was always there at the office when my hours ran late and I never felt alone because Ingolf was there, even on Saturdays and Sundays, and we would share some funny

tidbits about Indian and German culture, climate, and people. I also want to thank all my colleagues at the database group including Andreas Lübcke, Sandro Schulze, and Azeem Lodhi.

I met Sebastian Günther by coincidence. We immediately found out that we had common interests and our thought processes were much alike. Over the period, we shared numerous discussions, epiphanies, goals, and a lot of laughter. He has been a great colleague and a true friend. I want to thank him profusely for the fantastic collaboration we have had and one that we both hope to continue even if we go on separate paths post research.

This entire endeavor would have been too difficult had it not been for the support of my family. I want to thank my father, my mother, my brother and my sister-in-law, my nephew and all my relatives. My profoundest thanks go to my long-time girlfriend Radhika, who is now my wife. She supported me in incalculable number of ways and I thank her from the depth of my heart for being literally the better half of me.

Contents

List of Figures

List of Code Listings

List of Abbreviations

AST	Abstract Syntax Tree
DSL	Domain-specific Language
FOSD	Feature-oriented Software Development
IDE	Integrated Development Environment
JLS	Java Language Specification
SLOC	Source Lines of Code
SPL	Software Product Line
UML	Unified Modeling Language

Chapter 1

Introduction

1.1 Overview

Software engineering is the application of engineering principles to the development of software [Naur and Randell 1969]. Although intended to be a systematic, disciplined, quantifiable approach to software, software development process has been viewed as one fraught with perils. Software has certain characteristics that make applying engineering principles to it difficult; software is in general complex, it must conform to a variety of elements, it changes along many dimensions, and last but not the least, it is invisible [Brooks Jr. 1987].

Nevertheless, considerable progress has been made in taming the software beast in the four decades after the term *software engineering* first originated. This effort has been mainly driven by a set of fundamental principles that enable addressing the aforementioned *essential difficulties* involved in creating good software [McConnell 1999]. *Separation of concerns* is one such fundamental principle. It identifies the fact that creating a piece of software constitutes addressing many concerns and provides guidelines on how this can be achieved effectively and efficiently.

A *concern* is any semantically coherent issue in a given *problem domain* associated with a software system [Apel 2007]. It is any facet of a software system that requires special consideration. It can be anything specific such as persistence and concurrency [Hürsch and Lopes 1995] to a more general category such as functional concerns that are related to functionality provided to a specific client [Aldrich 2000]. While a problem domain consists of set of problems, the *solution domain* related to it consists of a set of solutions. An efficient treatment of involved concerns leads to a clean mapping between the problem domain and the solution domain [Czarnecki 2004].

This efficient treatment of concerns can be carried out by the principle of *separation of concerns* [Dijkstra 1976; 1982; Parnas 1976; 1979]. The separation

of concerns is centered around the idea that various concerns to be addressed by a software system should be separated or isolated and their representation should be untangled and localized. Concerns that are well separated can be *maintained* individually thereby reducing the overall complexity. It would be easy to *customize* them in isolation to conform to diverse requirements. An appropriate localization of concerns would present an opportunity to *reuse* them in changing situations. And finally, separated concerns would be easier to *comprehend*. It would lead first to a better understanding of individual concerns and eventually all concerns, enabling the developers to produce good software.

While the initial efforts to produce good software were helped by such advances as high level languages, time-sharing devices, and the use of an integrated development environments [Brooks Jr. 1987], the principle of separation of concerns offered a more fundamental solution at the level of design of a software system [Parnas 1976]. A software system would be designed keeping in mind important concerns and making allowance for their individual treatment [Parnas 1979]. Since the interface between a device on which a software runs and the design in the mind of developer is a programming language, the consideration of separation of concern led to two concepts namely, a specific *representation* of individual concerns in a given programming language and the *composition* of concerns thus represented.

Using the principle of separation of concerns, a software system could be *designed for change* [Parnas 1979] such that the developer could represent a concern by localizing the related code in a given programming language in a meaningful unit. Such a unit of code or a *module* would conform to a specification of *externally visible collective behavior of program groups* [Parnas 1972a;b]. This kind of *modular decomposition* of a software system targeting separation of concerns in a given programming language would enable the developer to combine and compose functionality addressed by individual concerns as per requirements. It would also enable him to create a family of software systems instead of a single system such that different members of this family contained pieces of functionality offered by the whole family [Parnas 1976].

While the concept of modules aided the advances in programming languages and practices such as *structured programming* [Dahl et al. 1972] and further advances in software design such as *structured design* [Stevens et al. 1974; Yourdon and Constantine 1979], the concept of program families led to better analysis and modeling of the problem domain of a software system [Kang et al. 1990]. A software system could have a number of stakeholders [Czarnecki and Eisenecker 2000], persons or entities interested in this software, each interested in a specific functionality within a problem domain. The requirements of stakeholders could be identified as features, individual members of family that contained specific features as software products, and the entire family as a *software product line* (SPL).

Software product line engineering (SPLE) thus emerged as a branch of soft-

ware engineering with focus on planning for and achieving product lines or families of software system instead of single systems, at the same time considering economic aspects [Clements and Northrop 2000; 2001; Czarnecki and Eisenecker 2000]. Many other software development paradigms can be used to develop SPLs such as, e.g., component-based software engineering [Szyperski 2002] and model-driven development [Clements and Northrop 2000; Czarnecki 2004]. A set of modeling techniques such as feature modeling [Czarnecki and Eisenecker 2000; Kang et al. 1990] and a set of implementation and programming paradigms such as feature-oriented programming [Batory 2004; Prehofer 1997] could be used that focused specifically on *features* as a kind of concern and their representation and composition to develop SPLs which have given rise to the development paradigm known as *feature-oriented software development* (FOSD).

While the phased-life cycle approach is also applied to SPLE with various phases trying to achieve *development for reuse* and *development with reuse* [Czarnecki 2004; Czarnecki and Eisenecker 2000], in FOSD all phases focus specifically on features as the de facto mechanism of reuse [Apel and Kästner 2009]. The roles of feature modeling and feature implementation techniques often cross the boundaries of phases for which they were initially intended. Feature models are often the product of analysis phase used for communication between stakeholders and at the same time they can be used to guide the product generation process [Apel and Kästner 2009]. Similarly, feature implementation techniques implicitly or explicitly refer to the feature model that underlies an SPL to obtain its products.

This dissertation aspires to make contribution to the ongoing research in this direction, i.e., how feature modeling and feature implementation can be best utilized in concert. It attempts to provide conceptual and practical answers to such questions as which kinds of models are needed in the overall development of SPLs? Is the model of an SPL sufficient or more models need to be mapped to the implementation? Which entities are important from the perspective of modeling and implementation of an SPL and at what level of granularity?

A multitude of feature modeling techniques for analysis and design of SPLs and feature implementation techniques for their implementation have been suggested for this purpose. The research on how to integrate techniques suggested for feature modeling and feature implementation continues to draw attention of researchers interested in leveraging feature models and feature implementation to the best of their abilities [Apel and Kästner 2009; Czarnecki 2004].

Even now the most desirable ways in which features can be modeled are being studied [Sinnema and Deelstra 2007] and how to best modularize features when implementing them is a topic of hot discussion [Lopez-Herrejon et al. 2005a]. This dissertation focuses on feature modeling and feature implementation and the relationship between them by considering the domains in which they operate, namely problem domain and solution domain respectively [Czarnecki 2004]. Instead of comparing individual techniques for modeling and implementing fea-

tures, we focus on features as a kind of concern and study its representation and composition in various techniques. Our analysis boils down to a set of requirements that when implemented enable integrating feature modeling and feature implementation while enabling the use of their core capabilities. The requirements thus obtained culminate in a representation and composition mechanism for features which we call **first-class features**. This dissertation presents an overview of our research efforts in the conceptual motivation behind first-class features, how we implement them, and our case-studies and observations that substantiate their merits.

1.2 Contributions

1. We present an analysis of which entities related to and including features are represented, how they are represented, and how they are composed in various feature modeling and feature implementation techniques. This analysis is preceded by a review of the current state of the art in the capabilities of features in modeling and implementation techniques from the perspective of problem domain and solution domain. The analysis itself is aimed at an evaluation of the relative merits which specific representations demonstrate and how these representations affect composition in each of the techniques.

2. Based on our analysis, we derive five requirements targeted at a new representation of features which we call **first-class features**. The implementation of first-class features would consist of an extensible and unified representation of modeling and implementation counterparts of features and feature models of an SPL and its products with first-class status in a host language with subsumed checking and identity retention. These requirements are coined in such a way as to be implementable in any given host language.

3. We implement first-class features in version 1.4 and version 1.5 compilers of static programming language Java using the JastAdd extensible compiler system [Ekman and Hedin 2007b;c]. This implementation is called **FeatureJ**. While FeatureJ is completely implemented by the author of this dissertation, the concept of first-class features is also implemented in dynamic programming language Ruby by a colleague in collaboration with the author. This implementation is called **rbFeatures**.

4. We demonstrate the practical applicability of using first-class features with four small to large size SPL case studies in each of FeatureJ and rbFeatures. First-class features prove advantageous due to cleanly separated semantics, clear and comprehensible syntax, implicit traceability links, ability for both composition and adaptation, and neatly achievable extensibility.

5. We provide a comparison of FeatureJ and rbFeatures in terms of how they are implemented. This comparison reveals the basic ways in which first-class features can be implemented and can be used toward implementing them in other host languages. It also throws light on the necessity of meta-level processing of different products of an SPL in terms of AST manipulation and metaprogramming mechanisms and indicates how compilation/interpretation models in the host language affect realization of various requirements of first-class features.

1.3 Structure

Chapter 2 lays the foundations for understanding the central ideas of this dissertation. Instead of providing exhaustive details of various software development paradigms, it focuses on describing the conceptual development within software engineering research, starting with the principle of separation of concerns, followed by representation and composition of concerns, leading eventually to SPLE and FOSD.

Chapter 3 discusses the the dual nature of features from the perspective of problem and solution domains. It presents the current state of the art in the capabilities of features classified by feature representation and feature composition. It then analyses various feature modeling and feature implementation techniques based on how features are represented and composed in each technique culminating into a set of requirements for *first-class features*. Finally, it discusses how the requirements of first-class features may be implemented.

Chapter 4 elaborates how FeatureJ is implemented atop Java 1.4 and 1.5 compilers using the JastAdd extensible compiler system. It first describes JastAdd's extensibility mechanisms followed by FeatureJ implementation specifics and finally its overall architecture using a running example of an SPL.

Chapter 5 elaborates how rbFeatures is implemented atop Ruby using Ruby's own syntax and semantics. It first describes Ruby's extensibility mechanisms followed by rbFeatures implementation specifics including its syntax and semantics and finally its overall architecture using a running example of an SPL.

Chapter 6 reviews the results of applying first-class features to four SPL case studies in each of FeatureJ and rbFeatures and reflects on how FeatureJ and rbFeatures conform to the requirements of first-class features based on our implementations and case studies.

Chapter 7 compares first-class features, first as they are implemented in FeatureJ and rbFeatures and then with various feature modeling and imple-

mentation techniques classified earlier in Chapter 3 into different categories based on the nature of feature representation and composition and presents merits and limitations of first-class features.

Chapter 8 summarizes the dissertation and its contributions, lists suggestions for further work, and puts forth perspectives for FOSD in general.

Chapter 2

Background

Go back to basics; remember the goals.

DAVID PARNAS
in the Keynote to Software Product Line Conference, 2008

In this chapter we lay the background for understanding the central ideas of the dissertation. Our attempt is to trace the historical origins of the concepts used in this dissertation and briefly indicate the conceptual development starting from separation of concerns leading to SPLE and FOSD.

2.1 Separation of Concerns

The term **software engineering** first appeared in the report of the NATO science committee conference in 1968 [Naur and Randell 1969]. In the ensuing years, many researchers and committees have opined about what software engineering means. One of the oldest definitions says that software engineering is the establishment and sound use of engineering principles to obtain software that works on real machines reliably and efficiently [Naur and Randell 1969]. The definition by Sommerville adds that software engineering is an engineering discipline which concerns itself with all stages of software development, from specification to maintenance [Sommerville 2004]. Bjørner combines these definitions and provides the following definition of software engineering [Bjørner 2006]:

> Software engineering is the establishment and use of sound methods for the efficient construction of efficient, correct, timely, and

> pleasing software that solves the problems such as users identify them, built by first understanding the problem domain and the user requirements and then designing the software to actually implement the desired solutions in a cost effective manner.

The software engineering movement as it were, was started in response to the perceived software crisis at the time [Naur and Randell 1969]. The term **software crisis** describes the mismatch between the rapid increase in computing power and the ability of the discipline of software engineering to deliver functional and efficient software on time. Specifically it addresses the perceived deficiencies in software projects that run over-time, over-budget and do not in fact meet requirements or prove inefficient and in some cases remain undelivered. The reports by the Standish groups tried to substantiate these claims over many years in their software failure reports [Glass 2006; Group 2003; Jørgensen and Moløkken-Østvold 2006; StandishGroupReport].

One may observe that software engineering is a process applied to obtain software that has specific characteristics: conformance to requirements, manageability and evolvability, cost effectiveness and efficiency. Good software would conform to the requirements of the users in order to serve its purpose. It would have manageable complexity so that it is easily maintained. It would enable reuse so that evolving requirements of users are addressed efficiently [Tarr et al. 1999]. Although the term engineering in software engineering implies the application of engineering to software, i.e., the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; there are some inherent differences between other engineering fields and software engineering. Brooks identifies these as the *essential difficulties* involved in creating good software. These are centered around the themes of conformity, complexity, changeability, and invisibility [Brooks Jr. 1987] as discussed below:

Conformity of software presumes that not only does a software need to conform to its specified usage, but that its easy mutability enables it to support extraneous demands. A software is created because someone or something (other software systems) needs it. Its interfaces must conform to the specific as well as changing demands of different users, increasingly involved scenarios, and increasingly evolving hardware technology. Much complexity in software arises because of the natural expectation that software can be made to conform any situation.

Complexity of software is apparent in the fact that no two parts of software are alike (at the level of statements in code). Scaling of software is not just a repetition of same elements in larger sizes rather it is an increase in the number of different elements and the complexity of the whole is more the sum of the complexity of parts. Technical problems such as product flaws, cost overruns, schedule delays, unreliability of software (because all possible states of the program are not understood), inability to easily

extend the software without side effects and management problems such as difficulty in maintaining the conceptual integrity of the software such that it functions as intended and inability to easily use other personnel to work with current software, can all be traced to the inherent complexity of software.

Changeability of software refers to the fact that the software of a system represents its function which is subjected to pressures of change along many dimensions. Successful software is tried beyond the confines of original intent and new uses are invented for it. While tangible objects are replaced by new objects beyond their normal life, a software is instead modified to fit new environment, evolved for new technologies, retrofitted with new requirements. It has been observed that the worst form of complexity results from software evolution [Nierstrasz and Achermann 2000].

Invisibility of software restricts the ways in which an exact specification of its function may be realized which will aid in verifying its correctness. The structure of software may constitute multiple views such as flow of control and flow of data and patterns of dependency [Brooks Jr. 1987] and it is difficult to coin a specification on the basis of which contradictions and flaws in the actual objects become immediately evident.

To enable software developers to create good software i.e., software that adheres to requirements of users, software that is of manageable complexity, software that can be evolved to cope with changes, and software that can be guaranteed to work as intended, many researches have suggested guiding principles. While many of these focus on characteristics of software development process such as phased life-cycle, continuous validation, and modern programming practices [Gould and Lewis 1985], iterative modify-test-modify cycle [Davis 1994], it is the principles which address the aforementioned difficulties inherent to software that are most vital [McConnell 1999]. One such principle known as **separation of concerns** stands out because of its pervasive use and its influence on software development methodologies since the time of its inception. In the following section, we take a review of this fundamental principle of software engineering and how it addresses conformity, complexity, changeability, and invisibility of software.

2.1.1 Separating the Concerns

The principle of separation of concerns was proposed in the 70's by Dijkstra [Dijkstra 1976; 1982] and utilized by Parnas [Parnas 1976; 1979]. In the paper in which he first presented the term separation of concerns [Dijkstra 1982], Dijkstra was referring to a way of thinking efficiently about any subject matter. A *concern* is a particular aspect of the subject matter under consideration [Dijkstra 1982]. Applied to software in general, it can be any issue related to the desired functionality of a software system. Dijkstra asserted that if one

were to address many aspects of the subject matter simultaneously, it would be difficult to do justice to all of them. Consequently, one should focus on a single aspect of the subject matter at a time.

Separation of concerns in software systems is achieved by separating, untangling, and localizing concerns in that system. It has been attributed to address the difficulties inherent to software in the following manner:

Customization to address conformity A stakeholder of software system is an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system [IEEE-1471-2000][1]. A piece of software can have many different stakeholders with specific requirements. In order to be usable for all stakeholders, i.e. to conform to various requirements, software system must be customizable. Customization means the adaptation of a software product to the needs of a particular audience [ISO/IEC-26514:2008]. Separation of concerns proves beneficial for this purpose because separated concerns can be adapted in isolation and combined as required.

Maintenance to address complexity Maintenance of software systems is the process of modifying the software system or a component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [ISO/IEC 2009]. With separation of concerns, the modification of software system can be perceived as the modification of implemented concerns. Complexity of software systems is the degree to which the system or a component has a design or implementation that is difficult to understand and verify [ISO/IEC 2009]. Separation of concerns into well defined entities at the implementation level enables finding potential flaws and their origins, since the concern implementation is isolated. Assuming that various concerns are well separated, maintenance activities can be carried out in a more orderly manner by modifying and testing required concern implementations in isolation.

Reuse to address changeability Reuse in software systems is the process of creating new software systems using existing software artifacts. Changes along various dimensions can be identified in terms of separated concerns. To the degree that the implemented concerns are isolated and independent, the process of affecting the required changes becomes easier because separated concerns can be more easily reused in different contexts and situations. Concerns separated into localized components can be modified independently and software can be built by different configurations of such components [Ernst 2003].

Comprehension to address invisibility When concerns are separated into a well structured system, it becomes easier for developers to understand

[1] We give references to specific standards from the *Software and Systems Engineering Vocabulary* at http://pascal.computer.org/ for some terms for which many sources exist. Software and Systems Engineering Vocabulary is a project of the IEEE Computer Society and ISO/IEC aimed at making authoritative definitions for software and systems engineering terms available from international standards.

them. Cleanly localized concerns can be treated in isolation from other concerns. This makes it possible to modify, adapt, reconfigure, and test concerns separately from one another. Apart from bringing order at the code level, well separated concerns also enable preserving the conceptual integrity of the software system, the functionality of which is represented by concerns at the implementation level.

2.1.2 Concerns

A **problem domain** of software system represents all issues of importance to be considered in order to create functional software. It is a set of similar problems that occur in an environment and lend themselves to common solutions [IEEE-1362-1998]. The environment where a solution or set of solutions resides is called the **solution domain** [IEEE-1362-1998]. The principle of separation of concerns states that various parts of software systems should be separated into meaningful units, which serve a singular purpose and enable addressing a specific issue of the problem domain. Every semantically coherent issue, the treatment of which should follow a certain set of behaviors, is known as a **concern** [Apel 2007].

A concern represents any facet of a software system that requires a special treatment. A concern can be anything from concurrency, persistence, failure recovery and exception handling to real-time constraints, distribution, and location control [Hürsch and Lopes 1995]. Multi-language support in software systems, platform-specific graphical look and feel, heterogeneous system policies are examples of some more concerns [Nierstrasz and Achermann 2000]. A higher level categorization of concerns is *functional concerns* such as piece of functionality provided for a specific client, *program organization* which represents the large scale structure of the program at the level of large interacting group of objects, *global and system-wide properties* that need custom code to be properly modified in specific situations, *repetitive code* such as invariant checking of a data structure or system of objects, logging of application behavior, tracing execution of large number of behaviors, consistent handling common errors throughout a common code base, *system performance* which refers to the performance of an entire subsystem of a program or load balancing of a number of system clients, and *context dependent behavior* [Aldrich 2000].

A broader classification of concerns is that in any software system, there are *basic concerns* and *special purpose concerns* [Hürsch and Lopes 1995]. A basic concern is responsible for supporting fundamental computational facilities and is used to implement the basic functionality. Special purpose concerns can be identified as fulfilling special requirements of an application, or managing and optimizing computational facilities provided by the basic concern.

Concerns are prevalent and pervasive in software systems. The commonalities in issues related to diverse software systems necessitates specific representation

and composition of concerns so that their implementation is well understood. The next section describes representation and composition of concerns in detail.

2.1.3 Representation and Composition of Concerns

The process of separation of concerns is achieved by *identifying, encapsulating*, and *manipulating* only those parts of software that are relevant to a particular concept [Ossher and Tarr 2000a]. The need for organizing and decomposing software into manageable and comprehensible parts necessitates a specific representation of concerns. The space of all relevant concerns in a software system is known as the **concern space**. A specific kind of concerns is referred to as a **concern dimension** [Ossher and Tarr 2000a]. Various concerns in concern space along the same dimension are one of a kind and share the same representation at the implementation level.

The process of *identification* of parts of systems that belong to a kind of concern is known as a **decomposition** mechanism. The parts of the systems that are identified as constituting a concern are then *encapsulated* using a specific representation at the implementation level and depending on the programming language used. For instance, with *functional* decomposition, a system is broken down into components that correspond to system functions and sub-functions. In an *object-oriented* decomposition, a system or component is expressed in terms of objects and connections between those objects.

Such decomposition of software systems along a single dimension results in what is termed as **tyranny of dominant decomposition**. Programming languages and methodologies tend to permit separation and representation of a single concern at a time. Encapsulation of basic concerns discussed earlier are examples of such dominant decompositions. For example, *classes* represent a basic concern which provides the basic functionality in an object-oriented programming language. *Functions* in functional languages and *rules* in rule-based languages similarly indicate basic concerns. Whenever a special purpose concern needs to be encapsulated and manipulated, in the absence of dedicated representation of these concerns, it becomes necessary to do so using decomposition mechanism available for basic concerns [Ossher and Tarr 2000a;b; Tarr et al. 1999].

In order to achieve the required functionality, the decomposed concerns representations must be combined using specialized **composition** mechanisms [Ossher and Tarr 2000a; Tarr et al. 1999]. The composition mechanism of separated concerns is directly dependent upon the representation of the separated concerns [Mulet et al. 1995]. For instance, object-oriented decomposition in which concerns are separated using classes and objects, enables composition of such concerns in following ways: (1) in the implementation part of an object, nested implementation objects are composed under the definition of encapsulating object, (2) composition of behavior takes place using inheritance and delegation mechanisms, and (3) interfaces enable creating set of protocols that

must be followed by each implementing class. The objects of such classes can be utilized in constructing complex structures [Aksit 1996]. When a system is developed using classes and objects as basic concerns, other special purpose concerns need to be represented in terms of classes and objects which results in using object-oriented composition for constructing software that addresses special purpose concerns. In other words, in the absence of specialized and dedicated representation of special purpose concerns, their composition is restricted by the composition abilities of the basic concern representations.

2.2 Family of Programs

A **design methodology** is a systematic approach to creating a design consisting of the ordered application of a specific collection of tools, techniques, and guidelines [ISO/IEC 2009]. Program families [Parnas 1976] a design methodology that enables creating a family of programs based on common properties. The concept of program families was motivated by the fact that if a set of programs are being developed, whose common properties are extensive, then it is beneficial to study the common properties of programs before analyzing them individually. This is further substantiated based on variations in application demands, variations in hardware configurations, possibilities to improve a pre-existing program based on alternate algorithms or application logic, and many experimental versions of a program [Parnas 1976].

The conceptual background for program families was laid out by Parnas [Parnas 1976] using two other design methodologies, namely *stepwise refinement* [Wirth 1971] and *information hiding module specification* [Parnas 1972a]. Using these methodologies, software is developed in incremental steps such that the principle of separation of concerns guides the design choices in each step of the development.

Parnas compared program family development using stepwise refinement and module specification with the classical sequential method of program development. He showed that both provide complementary advantages compared to the classical sequential method in which various complete programs are developed sequentially. Consequently, descendants of a given program share some of its ancestor's traits which can not be separated out even though deemed unnecessary because in the classical sequential programming, this would imply major re-programming [Parnas 1976].

In the following sections, we review how these two methodologies, namely stepwise refinement and module specification, apply the principle of separation of concern as the design principle to the development of families of programs.

2.2.1 Stepwise Refinement and Module Specification

To develop software based on the stepwise refinement methodology means the system is gradually developed in a sequence of *refinement steps* [Wirth 1971]. In each step, program elements such as data and instruction that operate on data, are decomposed and represented in terms of more detailed elements. This implies refinements to the specification of a program that are carried out until the elements are specified in terms of some programming language.

The specifications for instructions and data may be refined in parallel guided by design decisions based on an underlying criterion or a design principle [Wirth 1971]. Wirth states that programmers should be aware of the design principle and the alternative solutions that may exist (possibly due in part to the design principle used). Possible solutions can be envisioned as leaves of tree where each node basically indicates a decision guided by a design principle.

Parnas differentiated the method of information hiding module specifications [Parnas 1972a] from stepwise refinement in its ability to develop program families by stating that intermediate stages are not incomplete programs rather each of these node is a module which is a *specification of externally visible collective behavior of program groups*. Design decisions *not common to the members of program family* are identified and encapsulated into a *module*. The purpose of module specifications is not to make early decision about a program but to make it possible to postpone the decisions. As a result, module specifications enable relatively broad family compared to using stepwise refinement approach which progresses quickly to a narrow family based on limited variations due to early decisions.

Using module specification to define a program family, any of combination of set of programs that meets module specification can be treated as a member of the family. Similarly, module specifications can be parameterized to obtain family of specifications. A program based on a member of family of specifications is also a member of family of programs. Furthermore, programs consisting of set of programs based on subset of module specifications can be considered as members of a family.

2.2.2 Using Separation of Concerns as the Design Principle

Both stepwise refinement and module specification design methodologies are based on two concepts: representing an intermediate stage in program design and postponing decisions about actual implementation. When using separation of concerns to guide design choices in stepwise refinement, intermediate stages represent functionalities of the system. Actual implementations are added in stepwise manner to obtain a member of a program family. In order to use the principle of separation of concerns to guide the design choices in module specification, the criterion of *information hiding* [Parnas 1972b] is applied. A module is characterized by a design decision pertaining to a concern which it

hides from other modules which represent other concerns. The decomposition of system into modules uses another criterion which is *design for change* [Parnas 1979]. System details that are likely to change are treated as secrets of modules, i.e., changeable aspects of the system under consideration are the major candidates for modules [Parnas 1979; Parnas et al. 1984].

Toward the Concept of Software Product Lines

The core concepts related to separation of concerns and program families gradually evolved. Intermediate stages could be thought of as part of software system where functionality differed. The actual implementations could be selected to compose specific members of the program family. The engineering of program or system families pertaining to a specific application domain based on reuse was thought of as domain analysis [Gilroy et al. 1989; Prieto-Diaz 1987]. The concepts of concerns and design for change paved way to think of commonalities and variations in the family of systems. The common aspects of the domain and the differences between related systems in the domain were thought of as features that can be used to define mandatory, optional, and alternative characteristics of these related systems [Kang et al. 1990]. The concept of program families evolved into what is known as family-based software development or system-family engineering (SFE) [Weiss and Lai 1999] and later to software product line engineering (SPLE) [Clements and Northrop 2001].

2.3 Software Product Line Engineering

In literature, SFE and SPLE are used synonymously [Czarnecki 2004]. While SFE explores the commonalities among systems in a given problem domain, SPLE also concerns itself with scoping and management of products from an economic perspective. The Software Engineering Institute at Carnegie Mellon University defines a **software product line** (SPL) as follows:

> A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

In SPLE, software products that differ in functionality, called **variants**, can be rapidly created using reusable assets that can be anything from a common architecture to individual components [Czarnecki and Eisenecker 2000]. The ability of SPLs to change or customize an underlying software system is called **variability** [van Gurp et al. 2001]. A **variation point** indicates where the variation in the software system takes place [Jaring and Bosch 2002]. A variation

point is used to delay design decisions to a later phase in software development process such that each design decision constrains the number of possible variants [Coplien et al. 1998; van Gurp et al. 2001]. The concept of variation points clearly identifies the roots of SPLE to the separation of concerns and program families.

The engineering of SPLs is divided into two development processes: domain engineering (DE) and application engineering (AE). While DE is characterized as *development for reuse*, AE is identified as *development with reuse* [Czarnecki 2004]. We discuss each of these in the following sections.

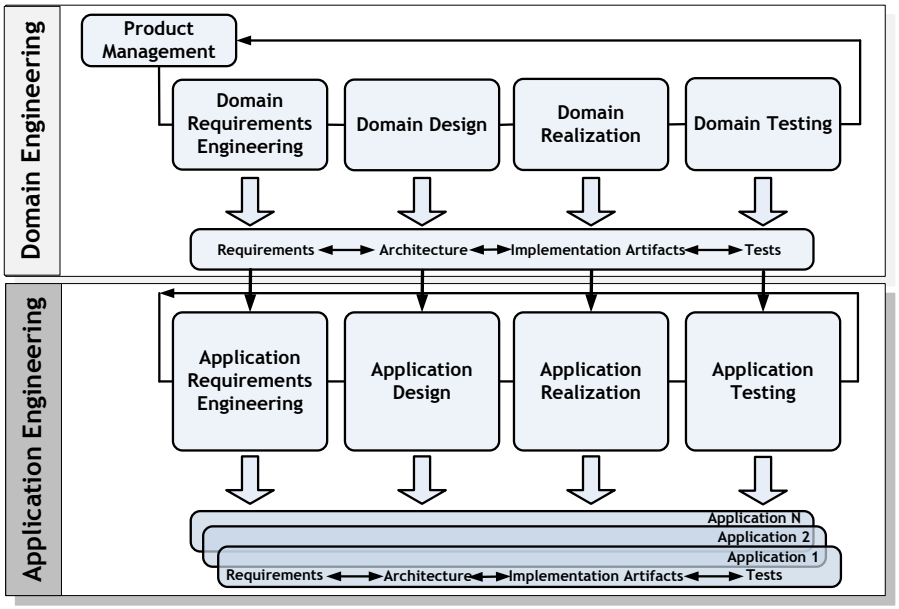

Figure 2.1: Domain and Application Engineering

2.3.1 Domain Engineering and Application Engineering

DE is also known as product-line development or core asset development. DE comprises four phases namely, domain analysis or domain requirements engineering, domain design, domain realization, and domain testing [Pohl et al. 2005] as shown in Figure 2.1:

Domain analysis or Domain requirements engineering consists of commonality and variability analysis. The sources of requirements are different stakeholders, existing applications, their documentation and failure reports. Commonality analysis determines which requirements are in fact

common to all applications. The specification of common requirements must maintain high quality because the work of different stakeholders depends on it. The goal of commonality analysis is to achieve as much commonality as possible [Ardis and Weiss 1997]. Following this, variability analysis is performed in order to identify variable requirements. The requirements that differ from each other indicate the need to introduce a variation point. Both common and variable requirements are documented using suitable notations [Kang et al. 1990] which include variability and constraint dependencies. In contrast to single system engineering, the commonality and variability analysis cater to the requirements of all envisioned SPL variants.

Domain design consists of producing main software structure and determine how common and variable requirements are reflected in it. It is used to create a common reference architecture for developing the members of the SPL that can be used to create individual product variants. Requirements that may conflict with each other are also reflected in the architecture. The reference architecture may be under-specified meaning that certain artifacts may not be addressed, e.g., artifacts that are variant-specific rather than being shared by all other variants of an SPL. In contrast to single system engineering, domain design tries to create architecture that can support mass customization.

Domain realization aims at providing detailed design and implementation of reusable software assets, based on the reference architecture obtained in domain design. It incorporates configuration mechanisms that will enable application realization to configure variants with the reusable assets. It distributes variability over reusable assets and also defines the binding times of variability such as whether the variation points are bound at compile-time, link-time, load-time, or run-time. In contrast to single system engineering, instead of building an executable variant, domain realization provides configuration and selection mechanisms for application realization.

Domain Testing consists of validating the realization artifacts and the output of other domain engineering sub-processes. The goal of domain testing is to ensure that requirements and design support testing. It aims at uncovering the evidence of defects in domain artifacts and creating reusable test artifacts for application testing. In contrast to single system engineering, domain testing faces the difficulty that no single executable configuration of reusable asset is to be tested and variability must be considered when performing testing.

AE is also known as product development. The main focus of AE is to use the reusable assets implemented in DE to build concrete applications. It consists of four phases namely, application requirements engineering, application design, application realization, and application testing [Pohl et al. 2005] as illustrated in Figure 2.1:

Application requirements engineering aims at eliciting and documenting the requirement artifacts for a particular application by reusing the domain requirements artifacts as much as possible. It consists of communication of domain requirement artifacts to stakeholders, specifically, customers. In contrast to single system engineering, application engineering must plan for external variability [Pohl et al. 2005]. It is possible that external variability requirements may arise leading to requirement deltas between application requirement artifacts and domain requirement artifacts which must be managed by application engineering. Furthermore, application engineering consists of documenting the traceability links between the domain requirement artifacts and application requirement artifacts. Finally, application engineering must create an application variability model based on variation points defined in domain variability model.

Application design consists of producing the application architecture. The application architecture is a specialization of the reference architecture developed in domain design. Given that reference architecture produced by domain design may be under-specified, application-specific artifacts may have to be designed by the application architect. In contrast to single system engineering, many application-specific requirements are specializations of domain requirements. Activities of application architect require less effort compared to single-system software and given that application-specific adaptations are not needed, the application architecture can be established by binding pre-defined variation points.

Application realization means developing applications that can be tested and brought to market. It consists of providing detailed design and implementation of application-specific components and configuring/selecting them to compose an application that is ready for testing. Contrary to single system engineering, the application developer selects reusable domain assets that conform to application architecture and builds an application by configuring application-specific components.

Application Testing is about achieving a sufficient quality of the application under test. It reuses the domain test artifacts and consists of unit and integration testing [McGregor 2001].The task of application testing is to validate the binding of variability and the configuration realized in the application. In contrast to single system engineering, application testing in SPLE must take into account the fact that application to be tested is created partly in domain engineering and partly in application engineering.

In addition to DE and AE, SPLE process also contains system tailoring. In system tailoring, application engineering delegates the new requirements, particularly those that are not covered by existing reusable assets back to domain engineering. The same steps of analysis, design, and implementation are applied to these new requirements and the implementation of reusable assets are updated so that they are capable of representing new requirements.

Benefits and Uses of SPLs

SPLs enable tactical engineering benefits which translate into set of strategic business benefits [Clements and Northrop 2000]. The tactical benefits include reduction in average time in creating and deploying a software product which implies improved productivity, reduction in average number of defects per product which results in increased quality, and reduction in average cost per product. These benefits translate into reduced time-to-market and time-to-revenue for new products, improved competitive product value, improved scalability of business model for given products and markets [Pohl et al. 2005; Van der Linden et al. 2007].

Dynamic Software Product Lines

In some domains, the underlying software requires extensive variation based on requirements as well as resource constraints, e.g., ubiquitous computing, medical and life-support devices, etc. The SPLs that are capable of *adapting* to changes in user needs and resource constraints are known as **dynamic software product lines** (DSPLs) [Hallsteinsen et al. 2008]. DSPLs are distinguished from SPLs by the fact that variation points are bound at run-time so that they can be adapted to the changes in the environment. Contrary to SPLE, engineering of DSPLs is not concerned with pre-runtime variation points. Instead of economic perspective, the ability to adapt to individual needs and situation is its main focus. It is also possible that different variation points are bound at different times including binding some variation points statically and others dynamically based on the domain-specific context [Alves et al. 2009].

2.3.2 Other Software Development Paradigms and SPLs

Many well-known software development paradigms can be used to create SPLs. In this section we take a quick review of these and state which characteristics differentiate them from SPLE.

Component Based Software Engineering The main goal of Component Based Software Engineering (CBSE) is to construct software systems on demand using off-the-shelf components. Szyperski [Szyperski 2002] defines components as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Components can be treated as reusable assets with which to build SPLs from [Clements and Northrop 2000]. They can be treated as units of software that can be combined to form software products as specified by the SPL architecture. Independent deployment can be interpreted such that components are installed into a product line's core asset

base where they are available for use in one or more software products. Such components must support the flexibility required to satisfy the variation points in the SPL architecture and/or SPL requirements. SPLE is sometimes mistakenly assumed to be another form of component-based development. While both component-based development and SPL development rely on the notion of components, SPLE consists strategic reuse and systematic and planned deployment of components, which are missing in component-based development [Clements and Northrop 2000].

Model-driven Development Model-driven Development aims at capturing all important aspects of a software system through appropriate models where a model is an abstract representation of system and its environment [Czarnecki 2004]. Models can be compiled into executable code and deployed at a customer's site. Model-driven approach to SPLs emphasizes automatic product derivation. The commonalities and variabilities in SPLs are expressed using models. The platform-independent models are translated into platform-specific models. Such models are used along with tool-supported translation to obtain the executable code of a product [Clements and Northrop 2000].

Many other methods for SPLs exist that support SPL development from architectural point of view [Matinlassi 2004]. These are distinguished as SPL architecture design methods. Examples include component-oriented platform architecture method for product family engineering (COPA) which is component-oriented but architecture-centric method for developing SPLs [Obbink et al. 2002], and Quality-driven architecture design and analysis (QADA) which aims at providing traceable product quality and quality assessment in developing SPLs [Matinlassi et al. 2002]. Our interest lies, on the other hand, in developing SPLs where the concept of features is of central importance, as we discuss next.

2.4 Developing SPLs Using Features

The development paradigm that focuses on the use of features for the construction, customization, and synthesis of SPLs is known as feature-oriented software development (FOSD) [Apel and Kästner 2009]. FOSD distinguishes itself from other methods of realizing SPLs in the pervasive treatment of features as the reusable assets. The basic idea behind FOSD is to decompose a software system in terms of features it provides and construct well-structured and customizable software using these features.

The concept of features is extremely involved. In simplest terms, it denotes a unit of functionality of a software system. Additionally, it may encapsulate a requirement, represent a design decision, and provide a potential configuration option [Apel and Kästner 2009]. The paradigm of FOSD itself emerged from different lines of research and this has resulted in multitude of definitions of

the concept of features. Before we take a review of these definitions, we quickly establish the background with the discussion of the concepts which are of primary importance to features, namely problem space and solution space and the mapping between them.

2.4.1 Problem Space and Solution Space

We have already seen the more general definitions of problem domain and solution domain (they are synonymous with problem space and solution space respectively)[2]. A problem space of software systems represents all relevant issues of importance to be considered and denotes a set of similar problems that occur in an environment related to a software system / software systems and lend themselves to common solutions. A solution space is the environment where these common solutions reside. We now revisit these concepts within the context of FOSD.

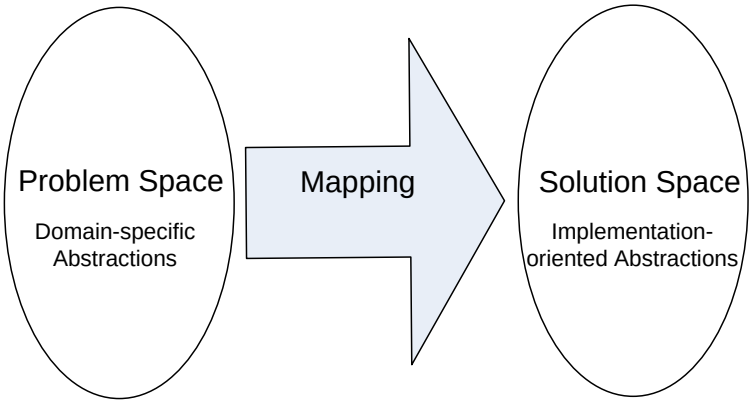

Figure 2.2: Problem Space and Solution Space [Czarnecki 2004]

The domain-specific abstractions in problem space enable a developer to adequately express the requirements in a manner natural to the domain. The problem space can be used to specify a desired variant from an SPL. The solution space defines how the requirements that conform to the intended behavior are realized. The solution space represents implementation-oriented abstractions with which to implement the requirements. A specific variant of an SPL

[2]The concepts of problem space and domain space were first used by Czarnecki and Eisenecker in the context of generative programming using components [Czarnecki and Eisenecker 1999] and later exemplified within the context of *generative software development* (GSD) paradigm. GSD is an SFE approach [Czarnecki 2004] and shares many similarities with FOSD except its stress on any implementation components or artifacts instead of features. The concepts of problem space and solution space are nevertheless applicable to FOSD as well and have been used for better conceptual understanding of FOSD [Apel and Kästner 2009].

can be generated by instantiating the related implementation-oriented abstractions.

Equally important as the problem and solution spaces is the mapping between them as shown in Figure 2.2. Within the context of SPLs, the mapping can be interpreted as a function that takes a variant composition specification as an input and outputs the implementation of the corresponding variant.

The domain-specific abstractions that describe the requirements of a software system and implementation-oriented abstractions that realize these requirements are known as features. We elaborate the concept of features in the next section.

2.4.2 Features

Features represent abstract concepts of a target domain and capture the requirements of software systems within that domain. At the same time, they also represent concrete artifacts which are composed together to actually realize the requirements. The abstract nature of features can be attributed to features being concepts in the problem space. The concrete nature of features can be attributed to features being realizations of these concepts in the solution space. When features are viewed only from one angle, it results in definitions of features that are more problem space-specific or solution space-specific. In the literature related to FOSD, features are predominantly treated as representing concepts from either the problem space or the solution space and this is reflected in the definitions offered for features. Many definitions exist for features. Below, we show definitions found in [Apel and Kästner 2009]. We divide these definitions according to the perspective of either the problem space or the solution space and re-arrange them according to when they were proposed in ascending order as follows:

1. From the perspective of the problem space, a feature is-
    a) a prominent or distinctive **user-visible aspect, quality, or characteristic** of a software system or systems [Kang et al. 1990]
    b) a **distinctively identifiable functional abstraction** that must be implemented, tested, delivered, and maintained [Kang et al. 1998]
    c) a **distinguishable** characteristic of a concept (e.g., system, component, and so on) that is **relevant to some stakeholder** of the concept [Czarnecki and Eisenecker 2000]
    d) a **logical unit of behavior** specified by a set of functional and non-functional requirements [Bosch 2000]
    e) a product characteristic that is used in **distinguishing programs within a family of related programs** [Batory et al. 2004]

f) a product characteristic from user or customer views, which essentially consists of **a cohesive set of individual requirements** [Chen et al. 2005]

g) a triplet, f = (R, W, S), where R represents the **requirements the feature satisfies**, W the **assumptions the feature takes about its environment** and S its **specification** [Classen et al. 2008]

2. From the perspective of the solution space, a feature is-

a) an **optional** [Zave 2003] or **incremental** unit of functionality [Batory 2004]

b) any **structure** that extends and modifies the structure of a given program in order to satisfy a stakeholders requirement, to implement and encapsulate **a design decision**, and to offer **a configuration option** [Apel et al. 2008b]

We have highlighted parts of each feature definition that bring forth important facets of the concept of features whether from the perspective of the problem space or the solution space. When developing a software system based on features, it is viewed as a collection of distinguishable characteristics specified in terms of abstractions that capture the intended functionality. These abstractions are implemented to realize the functional or non-functional requirements they identify. A family of products can be created (i.e., an SPL) and specific members (i.e., product variants) of this family can be composed using specifications of the same. A product is then a set of features and a variant of this product can be obtained by adding another feature incrementing the functionality offered by the product.

Features as domain-specific abstractions are modeled using feature models [Kang et al. 1990]. Feature models are the de-facto standard when it comes to graphically representing features in terms of requirements of a software system or distinguishable characteristics in a target domain. In the next section, we take review of feature modeling.

2.4.3 Feature Modeling

Feature modeling is a method and notation to elicit and represent common and variable features of the products in an SPL [Kang et al. 1990]. A feature model describes the relationships and dependencies of a set of features in a target domain [Czarnecki 2004]. Figure 2.3 illustrates a feature model of a text editor system [Czarnecki et al. 2005]. Such a feature model essentially denotes an SPL (in this case an SPL of text editors) and identifies the space of all possible variants that can be composed by selecting specific features from this model (in this case, individually configured text editors). Feature models are trees where the root of the tree represents the core concept in the system being

modeled. All nodes of the tree have parent-child relationship. The notations for the relationships between the children of the same parent indicate the ways in which these features can be selected if their parent was also selected in a variant.

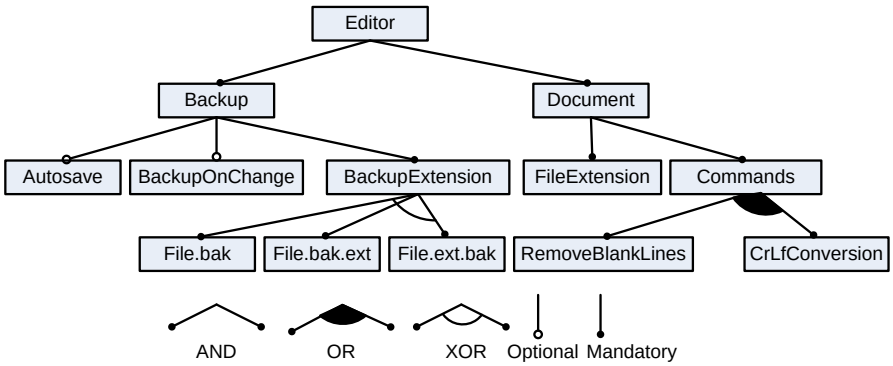

Figure 2.3: A Feature Model Representing a Text Editor Product Line

Feature models are central artifacts in FOSD. Over the time feature modeling notation has evolved to include many novel concepts [Czarnecki and Kim 2005; Czarnecki et al. 2002; 2004; 2005; Kim and Czarnecki 2005; Riebisch et al. 2002], e.g., the original feature model of the text editor shown above uses advanced modeling conepts [Czarnecki et al. 2005]. However, throughout this dissertation we refer to the original notation in [Kang et al. 1990]. A number of feature modeling techniques for engineering SPLs exist in which the use of feature models drive variant specification and generation [Asikainen et al. 2003; Becker 2003; Beuche et al. 2004; Boucher et al. 2010; Czarnecki et al. 2005; Deursen and Klint 2002; Loughran et al. 2008; Sinnema et al. 2004b;b]. They can be used to address different levels of abstractions including requirements, architecture and design, components and platforms, as well as different artifacts such as code, documentation, and other models, etc. The topic of feature modeling is one of the core topics of this dissertation and we return to it in the next chapter in detail.

2.4.4 Feature Implementations

Features are treated as concerns of primary importance in FOSD [Apel and Kästner 2009]. Beginning with Prehofer [Prehofer 1997], the need to explicate features in code has been realized using a variety of implementation mechanisms which contain different ways of representing features. Correspondingly, composition mechanisms are provided that enable composing features represented in specific ways. A bewildering array of methods, tools, and techniques exist for implementing features [Apel 2007; Apel and Batory 2006; Apel et al.

2005; 2009b; Batory 2004; Hundt et al. 2007; Kästner and Apel 2009; Kästner et al. 2007; Lai et al. 2000; Lopez-Herrejon et al. 2005a; McDirmid et al. 2001; Mezini and Ostermann 2004; Ossher and Tarr 2000b; Spencer and Collyer 1992; Xin et al. 2004]. The topic of feature implementations is one of the core topics of this dissertation and we return to it in the next chapter in detail.

2.4.5 Phases in FOSD

Similar to the two process model of SPLE, FOSD contains phases that concentrate on planning and developing families of products instead of single systems. Unlike the phases in SPLE, the phases of FOSD are more tightly interwoven and the thread that binds them is the concept of features. As discussed earlier, features are used pervasively in all phases of SPL development using FOSD. They are as follows [Apel and Kästner 2009]:

Domain Analysis in FOSD is based on feature modeling. It aim at identifying and capturing the commonalities and variabilities in the form of a feature model. As discussed earlier, from the initial conception, feature models have been extended in a variety of ways. This additional information is used later in the application configuration and generation process to rule out invalid product variants.

Domain Design is seldom treated as a separate phase in FOSD unlike its counterpart in the domain engineering process of SPLE. In contrast to domain design in SPLE, which focuses on creating a reference architecture, features are presumed to structure the design of the software system under consideration. Although first steps have been taken toward specification of structural and behavioral properties of involved features, the stress is given instead on using feature models as representing the structure where invalid requirements as well as required features are indicated through constraints on features in feature models.

Domain Implementation in FOSD aims at creating one-to-one mapping between the features identified in domain analysis to the features at the implementation level. Its focus is on recognizing code at varying level of granularities that captures the intended functionality of every feature in the feature model.

Application Configuration and Generation aims at achieving complete automation in generating specific applications from a software system. This is in contrast to its counterpart in SPLE, where stakeholders participate in assembly, integration, and adaption of reusable assets.

FOSD is a field of study with many challenges and a lot of active research [Apel and Kästner 2009]. We discuss these challenges in the next section.

2.4.6 Challenges in FOSD

As seen before, a number of techniques exist for feature modeling and feature implementation. Feature models are usable as communication artifacts and at the same time they can be used to drive the variant generation process. Feature modeling techniques use them to varying degrees. Similarly feature implementation techniques implement features in a variety of ways with some techniques being better at some facet such as than the others. Feature models represent both an SPL and a variant of that SPL when configured. Which kind of entities should be used in feature modeling and feature implementation is yet being explored in various techniques. Individual feature modeling and feature implementation techniques are yet difficult to integrate so as to provide a uniform perspective for all the phases of FOSD. This is the central challenge of FOSD which we address in this dissertation. In the next section we summarize this chapter and provide an outlook as to how we proceed in finding a balance of feature modeling and implementation towards a uniform FOSD technique.

2.5 Summary and Outlook

While software engineering is around four decades old (counting from the time of origin of the term [Naur and Randell 1969] to present), most of the FOSD-specific results have been obtained over last two decades (counting from [Kang et al. 1990] and [Prehofer 1997] to present). The origins of FOSD lie in the principle of separation of concerns, from which eventually the concept of program families evolved. It in turn gave rise to SPLE. Within the techniques used to engineer SPLs, those that focus on the concept of *features* are the basis of this dissertation. We take advice from Parnas stated in the epigraph to this chapter and go back to the concepts of concerns and their representation and composition. In the next chapter, we investigate how features and other entities are represented and composed in various feature modeling and implementation techniques and use this investigation toward the technique of *first-class features* in which we attempt to unify feature modeling and feature implementation.

Chapter 3

First-class Features

> Intelligence has two parts, which we shall call the epistemological and the heuristic. The epistemological part is the representation of the world in such a form that the solution of problems follows from the facts expressed in the representation. The heuristic part is the mechanism that on the basis of the information solves the problem and decides what to do.
>
> JOHN MCCARTHY
> *in Some Philosophical Problems from the Standpoint of Artificial Intelligence*

In this chapter we elaborate conceptual and concrete nature of features when viewed from the perspective of problem and solution spaces respectively. We review the current state of the art in the capabilities of features when modeling and implementing them as well as toward establishing traceability between conceptual and concrete features. We derive a problem statement toward achieving a unified perspective in FOSD and propose first-class features toward this end.[1]

3.1 Nature of Features

The two pivotal concepts in FOSD, namely feature modeling and feature implementation directly depend upon the domain-specific abstractions in problem space and implementation-oriented abstractions in solution space respectively. The aim of feature modeling is to provide methods and notations to elicit the common and variable features of an SPL targeted at a specific problem space

[1] This chapter shares material with the MCGPLE'08 paper *'Features as First-class Entities - Toward a Better Representation of Features'* [Sunkle et al. 2008b] and the technical report *'Representing and Composing First-class Features with FeatureJ'* [Sunkle et al. 2009].

[Czarnecki and Eisenecker 2000; Kang et al. 1990]. The aim of feature implementations is to provide mechanisms to represent features at the level of implementation in terms of solution space-specific platforms, languages, paradigms, etc. [Apel and Kästner 2009]. Features that are recognized as concepts from the problem space and implemented as entities in the solution space represent two sides of the same coin. We elaborate this in the following section.

3.1.1 Conceptual and Concrete Features

Features from the perspective of problem space denote concepts. The user-visible aspects, qualities, or characteristics [Kang et al. 1990], distinctively identifiable functional abstractions [Kang et al. 1998], characteristics distinguishable by stakeholders [Czarnecki and Eisenecker 2000], logical rather than physical units of behavior [Bosch 2000], and representation of a cohesive set of individual requirements [Chen et al. 2005] are all conceptual in nature. We refer to these features as the conceptual features. The conceptual features are often modeled using feature models.

On the other hand, features from the perspective of solution space denote concrete entities at the level of implementation. We refer to these features as the concrete features. Concrete features are code units that signify increment in functionality [Batory 2004; Zave 2003]. They are program entities or structures that extend and modify the structure of a given program [Apel et al. 2008b]. These features have an actual and concrete representation at the level of implementation. The concrete features can be represented by any code unit or program structure depending on the implementation language or a given programming paradigm. Henceforth in the dissertation, whenever we refer to conceptual features, we want to indicate features in problem space and whenever we refer to concrete features, we wish to indicate features in solution space.

The distinction between conceptual and concrete features and the resulting divide between feature modeling and feature implementation is the core problem of FOSD. As discussed in Chapter 2, the challenge is to provide a unified perspective for feature modeling and feature implementation. In order to clarify the distinction between conceptual and concrete features and how it divides feature modeling from feature implementations, we start by discussing the views on mapping between problem and solution spaces [Czarnecki 2004]. Our objective is to find where the division originates and what can be done to reconcile it.

3.1.2 Views on Mapping Between Problem and Solution Spaces

We have discussed earlier that problem space is a set of domain-specific abstractions and solution space is a set of implementation-oriented abstractions

[Czarnecki 2004; Czarnecki and Eisenecker 1999]. Equally important to the concepts of problem space and solution space is the mapping between them. The mapping acts a bridge between the domain-specific abstractions in problem space and implementation-oriented abstractions in solution space. The paradigm of FOSD depends on this mapping so that the composition specification of a variant of a software system that originates in problem space can be used to generate a specific variant in solution space. While the configuration view on mapping focuses on problem space, the transformational view on mapping focuses on solution space as explained next.

Configuration View

In the *configuration view*, problem space consists of domain-specific concepts which signify the features of a software system under consideration. The problem space defines the relations between the features and the dependencies/-constraints between the features. These help in sorting out valid and invalid combinations of features.

The mapping which provides variant composition and generation rules and the information about features, feature relations, and feature constraints combined together make up what is known as the *configuration knowledge* [Czarnecki 2004; Czarnecki and Eisenecker 1999]. It can be observed that the configuration knowledge is predominantly derived from the domain-specific abstractions in problem space as shown in Figure 3.1.

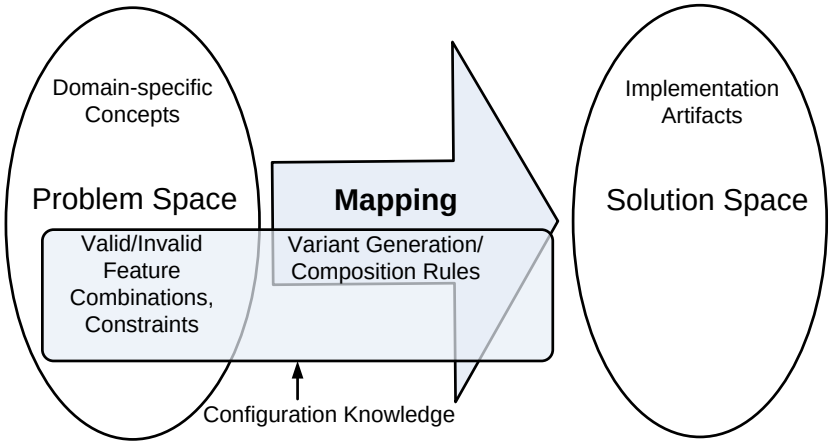

Figure 3.1: Configuration View on Mapping Between Problem and Solution Spaces[Czarnecki 2004]

The mapping between problem space and solution space provides a bridge between the features from problem space and implementation entities in solution

space. The input to the mapping between problem space and solution space is a specification of the given system such that specific features are selected and mapped to the implementation components in solution space. The output of the mapping is a composition of the implementation components that provides the functionality denoted by the selected features.

Transformational View

In the *transformational view*, problem space is represented by a domain-specific language [Czarnecki 2004]. A domain-specific language (DSL) offers expressive power targeted at a specific class of applications or facets of applications. When using a DSL to represent a problem space, the concepts of problem space are used as language entities. On the other hand, solution space is represented by an implementation language. The information about transforming DSL representation of features into the implementation components constitutes what we refer to as the *transformation knowledge*.

We specifically distinguish the transformational view from the configuration view by its focus on the concepts in solution space as opposed to focus on the concepts in problem space by the latter. The transformation knowledge then can be considered to be information about transforming any feature representations to the implementation components as shown in Figure 3.2. The mapping between the spaces can be considered to represent a transformation process that yields an implementation of a variant in a host language when input with a program with a specific representation of concrete features.

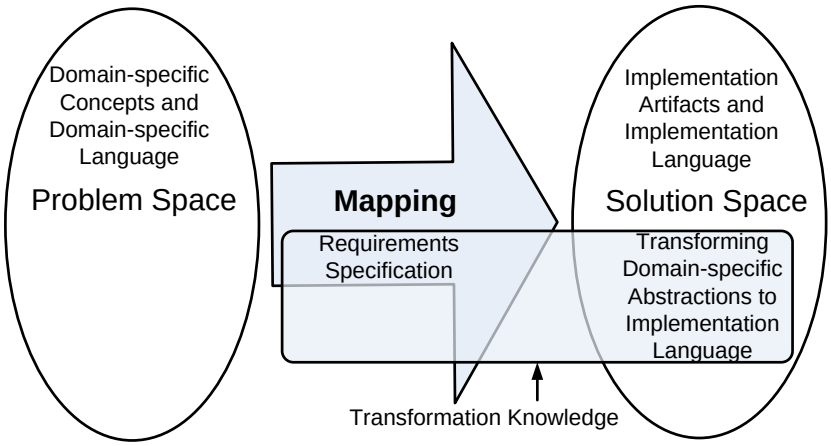

Figure 3.2: Transformational View on Mapping Between problem and solution spaces

Distinguishing Configuration and Transformational Views

Both configuration and transformational views use the mapping, but obtain the rest of the information required for the mapping to take place from problem space and solution space respectively. The configuration view on mapping can be assumed to take conceptual features as the starting point and maps them to implementation components. The transformational view on the other hand can be assumed to take the concrete representation of features as the starting point. It transforms the concrete features in the form of code units/program structures to implementation language entities.

The separation between problem space and solution space enables developers to structure the abstractions in them differently [Czarnecki 2004] which is the basic motivation behind having two different views on mapping between them.

The question remains as to what extent and with which characteristic capabilities are conceptual and concrete features represented and composed in feature modeling and feature implementation techniques and how they utilize the configuration and transformation knowledge. In the next section, we discuss the expected capabilities of conceptual and concrete features and how they relate to the configuration and transformational views discussed above.

3.2 Capabilities of Features - Current State of the Art

Many researchers have compared methods, tools, and techniques to represent conceptual and concrete features [Apel and Batory 2006; Apel et al. 2006; Lai et al. 2000; Lopez-Herrejon et al. 2005a; Mezini and Ostermann 2004; Sinnema and Deelstra 2007; Xin et al. 2004]. We are specifically interested in those studies that automatically have an FOSD perspective, i.e., studies that consider modeling and implementing features to engineer SPLs in contrast to studies that also include capturing variability in forms other than features. Furthermore, these studies should consider more than two techniques for feature modeling and/or feature implementation. Sinnema and Deelstra present a classification of modeling techniques for conceptual features [Sinnema and Deelstra 2007]. Lopez-Herrejon et al. evaluate support for concrete features in terms of various modularization technologies [Lopez-Herrejon et al. 2005a].

The methodology used by both these case studies is as following:

1. A particular problem is defined. In case of classification of feature modeling techniques, the problem is defined as an SPL in the domain of license plate recognition on handhelds devices with different operating systems and camera interfaces [Sinnema and Deelstra 2007]. In case of evaluation of feature modularization techniques the problem is an SPL version of the expression problem [Lopez-Herrejon et al. 2005a].

2. A set of properties are specified. These properties are used as *categories of classification framework* [Sinnema and Deelstra 2007], or that can be *readily inferred from, illustrated by, and assessed* [Lopez-Herrejon et al. 2005a] with respect to the defined problem.
3. Each feature modeling technique or a feature implementation technique is evaluated and classified based on how it supports a given property while resolving the defined problem.

The most notable aspect of these classification and evaluation studies is that they bring forth capabilities and desirable properties of features, both conceptual and concrete as elaborated next.

3.2.1 Toward Modeling Conceptual Features in Problem Space

The techniques that aim at representing conceptual features in problem space share similarities and differences in terms of core characteristics important in terms of modeling concepts. When applying such techniques two aspects are important to developers: what it is that they want to model and what constructs are available to do so and what kind of support is provided in creating and using these models. These aspects are used to classify feature modeling techniques toward how effectively they model the conceptual features in problem space [Sinnema and Deelstra 2007] as enlisted below:

Representation of Conceptual Features

- *Ability to Choose* - Variability is about choices that enable developers to change, configure, customize, and extend product family artifacts for use in a given context.
- *Representation of Product/Variant Model* - A product/variant model refers to how selection/configuration of features available in problem space is represented in the feature modeling technique.
- *Constraint Representation* - This signifies the ability to test valid variants based on the inclusion and exclusion constraints amongst the conceptual features in problem space.

Composition of Conceptual Features

- *Abstraction Levels* - This denotes how variability is modeled over different abstraction levels such as features, architecture, and implementation and what kind of transition is made available between the levels.
- *Effectuation* - The mechanism to actually obtain a product variant is referred to as *effectuation*.

Further characteristics that would be helpful but are not absolutely necessary include specification of quality attributes such as functional/non-functional

properties, support of incompleteness/imprecision in information and specification and configuration guidance (especially important in GUI-based feature modeling tools) [Sinnema and Deelstra 2007].

Note that the representation capabilities of conceptual features largely draw upon the configuration knowledge described earlier in the description of the configuration view on mapping between problem and solution spaces. The ability to choose, configure, and customize a representation of product/variant model, constraint representations etc. is what the configuration knowledge consists of. Feature modeling techniques use this knowledge toward generating variants. This indicates that the stress in feature modeling techniques is on using configuration knowledge rather than transformation knowledge. The capabilities of concrete features on the other hand, mostly refer to transformation knowledge as we discuss next.

3.2.2 Toward Implementing Concrete Features in Solution Space

The techniques that aim at implementing concrete features in solution space have very different notions of how concrete features are modularized and how they are represented and composed. The properties of feature modularity important for developers are how are features represented in the code and how are they composed [Lopez-Herrejon et al. 2005a] as enlisted below:

Representation of Concrete Features

- *Program Deltas/Code Fragments* - This refers to the ability to capture code fragments of wide range of granularity in the given programming language.

- *Naming and Identification* - This is the ability to identify the code fragments with the name of the feature to which they belong, so that they can be referred to and manipulated.

Composition of Concrete Features

- *Flexibility* - This indicates that feature modules should be syntactically independent from the rest of the code of an SPL such that it is not hard-coded according to a specific composition. Flexible composition property enables reuse of feature modules by enabling them to participate in any valid composition.

- *Order Independence* - This signifies the ability to obtain a variant without being tied up to a specific ordering of features when composing them.

- *Static Typing* - It should be possible to statically type check the composition of a variant to ensure that it is a valid variant (based on feature relations and constraints) and it can compile without errors.

Note that the capabilities of concrete features do not explicitly refer to configuration knowledge. What is given importance instead is the ability to transform a concrete representation of features capturing code fragments from a wide range of granularity to the host language representation in order to obtain a variant. This indicates that in case of feature implementation techniques, the stress is more on transformation knowledge rather than configuration knowledge.

Until now, we discussed the capabilities of feature modeling and implementation techniques in representing and composing conceptual and concrete features respectively. We have seen that feature modeling and feature implementation techniques derive most of the required information from problem space and solution space in terms of configuration knowledge and transformation knowledge respectively. There is another aspect that directly depends on the mapping between the spaces and on both configuration and transformation knowledge known as **traceability** which we elaborate next.

3.2.3 Toward Establishing Traceability between Conceptual and Concrete Features

Traceability is defined as a discernible association among two or more logical entities, such as requirements and system elements and represents the degree to which requirements and design of a given system element match [ISO/IEC 2009]. A more detailed definition of requirements traceability is that it is the ability to describe and follow the life of a requirement, in both forward and backward directions [Gotel and Finkelstein 1994], i.e., from its origins, through its specification and development, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases.

In Section 2.3.1, we discussed that it is the responsibility of application engineers to document the traceability links between domain requirements artifacts and application requirement artifacts. At the same time, it has been established in the literature that variabilities at different levels of abstraction and across different development phases are associated with each other and that they need to be linked to simplify evolution and maintenance of an SPL [Berg et al. 2005]. While approaches have been suggested that enable traceability links between conceptual and concrete features using additional models and tools [Beuche et al. 2004; Sochos et al. 2006], these only consider the traceability between a feature and an artifact [Berg et al. 2005].

Lago et al. present an approach to traceability management in which they identify core traceability paths between feature modeling and feature implementation [Lago et al. 2009]. Note that *feature implementation* that we have

shown in Figures 3.3 and 3.4 is referred to as *structural models* by Lago et al. Their terminology is based on the fact that whereas feature models denote domain-specific properties, the structural models reflect architectural decisions and basically identify implementation components. While we retain the general idea behind various traceability paths, we chose to refer to the structural models as feature implementation, keeping in mind the FOSD perspective.

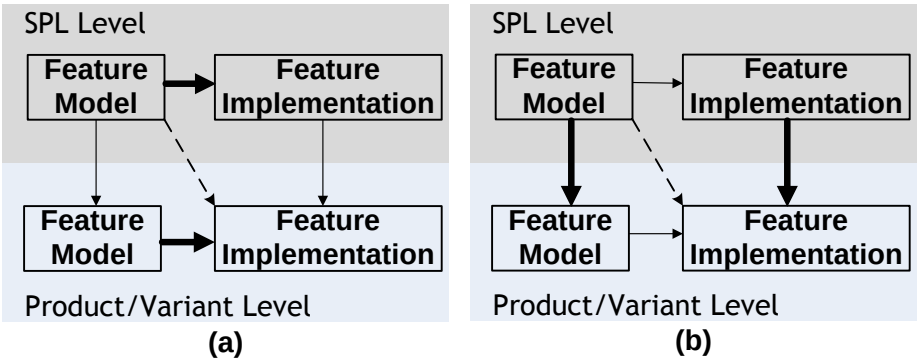

Figure 3.3: Traceability in SPLs - I

The traceability paths identified by Lago et al. are: from the conceptual features to the concrete features, from a feature model to a product/variant model, and from the implementation of an SPL to the implementation artifacts and from the implementation of a product to the implementation artifacts [Lago et al. 2009]. We elaborate these in the following:

1. ***Traceability between conceptual and concrete features*** This type of traceability is required when we need to understand the impact of a change to an SPL. An example is when a feature is removed, we need to know the affected entities and this information must come from the feature model (to evaluate impact on dependencies between conceptual features) as well as the feature implementation (to evaluate how removal of a feature affects the implementation of other concrete features). This is shown in Figure 3.3-(a).

2. ***Traceability between a feature model and a product/variant model*** The second type of traceability indicates the need for explicit links between the SPL level models and the variation decisions taken for each product. Such traceability (illustrated in Figure 3.3-(b)) is required to be able to ascertain whether the properties of an SPL are reflected in a product [Lago et al. 2009].

3. ***Traceability between SPL and Product Code and Implementation artifacts*** This type of traceability identifies the need for explicit links between the application code and code that makes up an SPL and

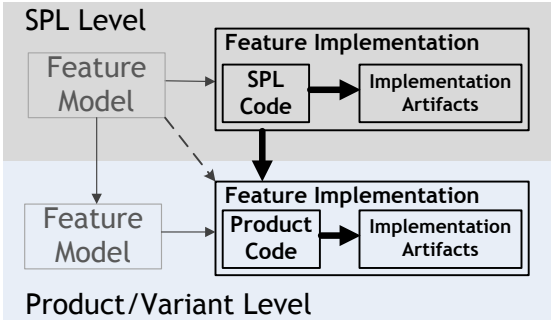

Figure 3.4: Traceability in SPLs - II

a product of that SPL as shown in Figure 3.4. This kind of traceability is particularly meaningful in case of legacy applications that are to be converted to SPLs, e.g., when generating variants from an application by overlaying an SPL architecture on it. Furthermore, during reuse it is often necessary to recognize products that are compliant with certain architectural style or coding and other standards [Lago et al. 2009].

Note that the dashed arrow in Figures 3.3 and 3.4 from feature model at SPL level and feature implementation at product/variant level indicates the derivation of a specific product/variant from an SPL [Lago et al. 2009]. In our opinion, realizations of these traceability paths indicated in Figures 3.3 and 3.4 signify the desirable properties that feature modeling and implementation techniques should have in order to establish traceability between feature models and feature implementation. We illustrate this in what we call the *traceability view on the mapping between problem and solution spaces* shown in Figure 3.5.

Notice that in Figure 3.5 we explicitly connect problem and solution spaces instead of using unidirectional mapping to establish traceability. Furthermore, we specifically wish to indicate the fact that in order to achieve traceability along with proper modeling and modularization of features both configuration knowledge in problem space and the transformation knowledge in solution space are required.

3.2.4 Problem Statement

In the preceding sections, we discussed the desirable properties of feature modeling and feature implementation techniques in modeling variability in problem space and modularizing it in solution space. Along with the representation and composition properties of conceptual and concrete features, we also discussed the traceability properties these techniques should possess. Based on this discussion, we make the following observations:

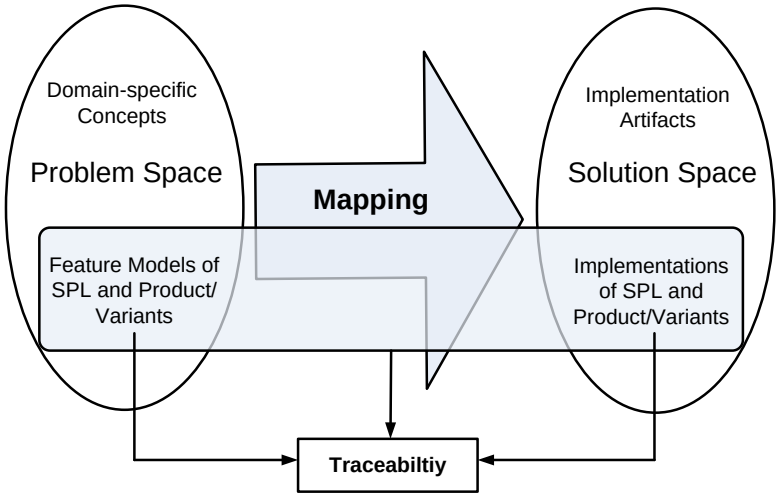

Figure 3.5: Traceability View on Mapping

1. The feature modeling techniques for developing SPLs focus on the conceptual features. The dominant view shared by these techniques is the configuration view, since the information used by these techniques is derived mainly from the configuration knowledge. These techniques use representations of conceptual features, variation points, feature models, product/variant models, and constraints. We refer to these as the *feature modeling entities*.

2. The feature implementation techniques for developing SPLs focus on the concrete features. The dominant view shared by these techniques is the transformational view, since the information used when modularizing and composing concrete features is derived mainly from the transformation knowledge.

3. The traceability between problem space and solution space depends on both conceptual and concrete features. Additionally, establishing traceability links between feature models and feature implementation across the SPL and the product levels requires representation for feature models, product/variant models, and constraints at the level of implementation apart from the concrete features. Furthermore, SPL code and product code needs to be identified separately from the implementation artifacts. We refer to these entities which are required to have their own representation or a means of identification at the level of implementation as the *feature implementation entities*.

We refer to the feature modeling entities and feature implementation entities collectively as the **feature domain entities**. We also observe that the desirable

properties of feature modeling techniques do not consider implications at the level of implementation and vice versa. Similarly, both ignore the implications toward the traceability properties. Furthermore, all of the case studies mentioned so far conclude that no single technique satisfies all properties [Lopez-Herrejon et al. 2005a; Sinnema and Deelstra 2007]. Instead, various techniques support the respective desirable properties to varying degrees.

We also observe that some representations of conceptual and concrete features prove better at solving a specific problem related to FOSD. For instance, Mezini and Ostermann compare the abilities of feature and aspect-based modularization mechanisms with respect to the problem of requesting stock information and discuss aspectual collaborations to tackle problems such as hierarchical features, lack of reuse, and lack of support for composition at run-time faced by refinement-based and aspect-based techniques [Mezini and Ostermann 2004].

Other comparative studies focusing on a smaller set of techniques similarly show the strength of a particular representation and composition mechanism for concrete features over the other [Apel and Batory 2006; Lai et al. 2000; Xin et al. 2004]. Other researchers compare various techniques based on *the ease with which the set of desirable properties can be supported* by a given representation and composition mechanism for concrete features. For example, while it is shown that combined aspects-refinements-based representations are better at handling concrete features that crosscut the application code [Apel et al. 2006; Mezini and Ostermann 2004], it is also discussed that approaches in which features are represented as annotations score over approaches that modularize and compose features in terms of ease with which crosscutting features are handled [Kästner and Apel 2009]. It is possible that even though all the desirable properties of feature modeling or feature implementation were supported by two techniques, one technique scores more over the other.

We argue that in order to achieve a unified perspective for FOSD and provide support for modeling, modularizing and implementing, and achieving traceability, we need to investigate the core characteristics of representation and composition of features that enable supporting the desirable properties of respective techniques in the first place. We need to find the correct level of abstraction for feature domain entities in these representations and reconcile the focuses on either problem space or solution space. Toward this end, we formulate the following research questions:

1. Which of the feature domain entities should be represented and what kind of representation of feature domain entities is required toward a unified perspective?
2. What level of abstraction would enable support for the desirable properties from modeling and modularization perspective as well as toward establishing traceability?

The first question targets the fact that different techniques provide explicit representation of some feature domain entities but vary in terms of which feature domain entities are represented and how they are represented. The second question aims at finding the level of abstraction of feature domain entities represented explicitly that proves to be suitable toward combining feature modeling and feature implementation at the same time providing opportunity for traceability between the two. In the following, we describe how we intend to approach these questions.

3.2.5 Proposing a Solution

We approach the solutions to these questions in three steps outlined below:

1. ***Analysis of Feature Representation and Composition in Various Techniques*** We present an analysis of feature modeling and feature implementation techniques targeted at finding the core characteristics of representation and composition of feature domain entities. We also indicate whether these representations explicitly consider traceability properties discussed before.
2. ***Proposing New Representation and Establishing its Requirements*** Based on this analysis, we propose a representation of feature domain entities at the level of abstraction we deem to be most suitable for supporting feature modeling, modularization, and traceability properties. Furthermore, we establish a set of requirements in order to be able to implement and test such a representation and accompanying composition mechanism.
3. ***Implementing the New Representation and Evaluating with Case Studies*** Based on these requirements, we implement the new representation of feature domain entities and evaluate it by applying it to various scenarios related to creating and maintaining SPLs.

In the next section, we present the analysis of feature representation and composition in various modeling and implementation techniques as the first step toward the solution.

3.3 Analysis of Feature Representation and Composition

In the following, we analyze and differentiate techniques based on the primary kind of representation of feature domain entities. We then state which feature domain entities are represented and how. In order to simplify the analysis, we first categorize feature modeling techniques followed by feature implementation techniques.

3.3.1 Feature Modeling Techniques

The feature modeling techniques used for managing variability in problem space basically differ based on the properties specified earlier in Section 3.2.1 on page 32. The ability to choose is presented as either *multiplicity in structure* in which choice is made between features or as *choice models* in which choice is made between variation points instead of individual features. The product/variant models are represented either as stand-alone entities without any link to choices made or as *decision models* in which product/variant models are defined in the context of features/choices in the variability model. In both cases they are *feature description profiles* in some format such as XML. The constraints in variability models are represented as inclusion/exclusion constraints or as algebraic expressions specified using an external constraint language. The abstraction levels are arranged either in a hierarchical manner or in multiple layers to manage large number of features and constraints. The product/variant models created during configuration indicate descriptions of which artifacts should be present in the final application. The XML or other formats of description are transformed to set of artifacts that represent the final product.

Based on an already available classification [Sinnema and Deelstra 2007] discussed above, we now distinguish various feature modeling techniques in terms of representation and composition of feature domain entities.

1. **GUI-based Feature Models**

   These are the techniques that represent various aspects of feature modeling such as the features/variation points, the product/variant models, constraints amongst features as *graphical* entities. Examples include CBFM (cardinality-based feature modeling) [Czarnecki et al. 2005], COVAMOF (ConIPF (Configuration in Industrial Product Families) Variability Modeling Framework) [Sinnema et al. 2004b], and Pure::Variants [Beuche et al. 2004]. While CBFM and pure::variants are implemented as an Eclipse plugins, COVAMOF is implemented as a set of Visual Studio plugins.

   Feature Representation

   While the conceptual features are represented as graphical entities, there is no explicit representation of concrete features. Pure::Variants and CBFM use multiplicity in structure while COVAMOF uses choice models to represent features and SPLs as modeling entities. Products are represented as decision models i.e., a variant is defined within the context of an SPL.

   Feature Composition

   The descriptions of product/variant models are obtained in textual format such as XML and used to compose artifacts in solution space. While

Pure::Variants uses preprocessor mechanism at the level of implementation to include code belonging to concrete features, COVAMOF's description profiles consist of actions to be taken in order to obtain a variant by composition of components [Sinnema et al. 2004b].

Traceability

Out of these techniques pure::variants provides a connector (termed an *addin*) to IBM Rational DOORS® which is a family of requirements definition and management solutions, aimed at supporting traceability between requirements selected in a variant [Systems 2010], i.e., traceability between feature model to a product/variant model. COVAMOF provides what is called as COVAMOF Variability View which can be used for traceability [Sinnema et al. 2004a]. There is no explicit consideration of traceability in CBFM [Czarnecki et al. 2005].

2. **Features as Language Entities**

These are techniques that represent some or all of modeling entities in textual format such as a domain-specific language. In these approaches, the use of language representation of modeling entities does not pervade all aspects of modeling specification and product generation. Examples include Variability Specification Language (VSL) [Becker 2003], AMPL (Asset Model for Product Lines) modeling language [Sinnema et al. 2004b], Feature Description Language (FDL) [Deursen and Klint 2002], and Text-based Variability Language (TVL) [Boucher et al. 2010].

Feature Representation

The conceptual features are represented in terms of feature definitions and operations at the level of a DSL. Various algebraic transformations may be applied to features as language entities. The concrete features again do not have an explicit representation. For example, VSL uses a general meta-model of variability along with preprocessor inspired XML based language descriptions. AMPL is used internally in COVAMOF modeling to represent a product model in terms of mandatory and optional features, feature parameters (selected from graphical front-end) and restrictions (i.e., constraints). FDL represents all modeling entities as DSL constructs and proposes feature diagram algebra to be used with these constructs. TVL proposes to represent feature modeling entities with C-like syntax [Boucher et al. 2010].

Feature Composition

The definitions of conceptual features have been proposed to be mapped to artifacts such as Unified Modeling Language (UML) models using an intermediate format (usually in XML) that transforms feature descriptions to aggregation, associations, subclasses relations in UML [Becker 2003; Deursen and Klint 2002]. The UML models are used in generating classes that represent the final application. The generated product/variant has no traceability link to the original feature model representing the

SPL. Feature composition has not been explored in VSL, FDL, and TVL as all of them are at the proposal state while AMPL is used internally and is not vital for composition of components.

Traceability

Out of these techniques only VSL considers traceability as a desirable characteristic [Becker 2003], the rest of the techniques do not consider any traceability properties described earlier.

3. **First-class Architectural Variability**

The feature modeling languages in this category differ from the second category stated above by the fact that not only the use of modeling entities as language constructs pervades all stages of modeling including specification of SPLs and product models but the connection to implementation components are specified using special constructs in the language. Examples of this type of modeling languages include Variability Modeling Language (VML) [Loughran et al. 2008], and Koalish [Asikainen et al. 2003].

Feature Representation

The languages which enable first-class representation of architectural variabilities represent all modeling entities as language constructs. In VML e.g., an SPL, a variation point, a feature, constraints between features as well as variation points are represented as language entities. Furthermore, it contains references from variation points and features to actual architectural components. Koalish language incorporates variability into Koala components. While Koalish does not explicitly represent features in terms of variation points and concrete features, like VML it contains references to Koala components and establishes an explicit connection between modeling and implementation via language constructs.

Feature Composition

In these techniques, composition of implementation components is carried out using references from the modeling entities expressed in the languages. Furthermore, implementation components are components rather than features. While VML uses the concept of references and actions [Loughran et al. 2008], which act as component activation and component composition decisions, Koalish uses Kumbang configurator and generates Koala description of products which is used to generate C source files (a combination of make files, and .c and .h files) that can be compiled into an executable.

Traceability

While developers of VML do not consider traceability originally [Loughran et al. 2008], they are considering the implementation of *trace-links* [Zschaler et al. 2009], but which traceability properties will be considered is unclear. Koalish itself does not provide traceability support but the Kumbang configurator used by it addresses traceability from requirements to

architecture and architecture to components [Asikainen et al. 2003], i.e., traceability properties (a) and (b) shown earlier in Figure 3.3 on page 35.

We see that there is a progression from representing feature modeling entities as graphical entities, to representing some of the modeling entities at the level of language, to representing all modeling entities as first-class language constructs. FDL, VSL, and TVL are based on the premises that (1) a language representation is amenable to automatic processing of various kinds, (2) a language representation enables managing variability in a uniform manner, and (3) equivalent textual notations in the form of dedicated language constructs enable scalability for feature models as well make them understandable to most stakeholders.

We support these premises and approve the uniformity of notation, since using a language representation it becomes possible to position both the modeling and the implementation entities at the same level of abstraction. At the same time, these techniques do not achieve what we believe to be another important advantage of language representation which is to establish an implicit traceability link to implementation components. In case of VML, this is achieved to some extent using explicit references from the modeling entities to variation points. Even in VML, the implementation entities are components rather than features. Koalish achieves traceability to some extent but like VML implementation artifacts are components in Koalish rather than features.

Based on this analysis, we conclude that an explicit representation of feature modeling entities at the level of language offers better opportunities than a graphical representation towards achieving a unified perspective if instead of components, feature implementation entities were represented at the level of language along with feature modeling entities.

3.3.2 Feature Implementation Techniques

The techniques used for modularizing variability in solution space vary based on the properties specified in Section 3.2.2. The fact that a specific code fragment/program delta of given granularity belongs to a concrete feature is expressed either using preprocessor directives, annotations, or as a separate module that contains the said fragment based on the underlying modularity mechanism/programming language. Static typing support is generally based on type-checking provided in the underlying integrated development environment (IDE) such as Eclipse, or the type-checking in underlying programming language. Dependence on order is based on whether features are added incrementally or all at once such as in techniques that represent concrete features in separate modules or as annotations respectively. In techniques in which order of composition of concrete features cannot be determined a-priori, mechanisms such as precedence operators may have to be used to indicate correct order.

Based on an already available classification [Lopez-Herrejon et al. 2005a] discussed above, we now distinguish various feature implementation techniques in terms of representation and composition of feature domain entities.

1. **Features in terms of Preprocessors and Annotations**

   In these techniques, the code belonging to different features is put inside preprocessor directives or annotations and added to the source to be compiled depending on the features selected. Examples of such techniques include #ifdef preprocessing [Spencer and Collyer 1992] and virtual separation of concerns that uses colored annotations over Java code fragments in the Eclipse IDE [Kästner and Apel 2009].

   Feature Representation

   While code fragments of a concrete feature are identified in terms of directive identifiers or annotation names, there is no representation of conceptual features in techniques using preprocessor directives.

   The virtual annotations approach provides support for feature models to which virtual annotations (that is Eclipse editor-based background colors) are mapped and consistency is maintained between features in the feature model and the virtual annotations, so that ad-hoc coloring of code fragments is avoided. A maximum of 12 colors are repeatedly used so that it is possible that multiple features are represented by the same color [Kästner 2010]. Furthermore, modularity for concrete features is *emulated* so that all code fragments belonging to a feature can be viewed such that rest of the code is hidden in a Eclipse view called "views on a feature". Similarly, a product/variant model is emulated in terms of "views on a variant" such that except the code fragments that belong to selected features, the rest of the code is hidden [Kästner 2010]. The views are editable in the sense that changes made to code fragments in a view reflect in the original source code.

   Feature Composition

   The code belonging to different features is added to the source to be compiled depending on the features selected in a variant. Upon compilation a specific variant is obtained. The composition is guided using configuration scripts and other textual formats to indicate all the available features as well the features selected in a variant.

   In case of virtual annotations, colors are interpreted as conditional compilation instructions [Kästner 2010] and code of a particular variant is kept for compilation. While emulation of concrete features and variants provide visual aid to a developer aimed at better comprehension of multiply colored code fragments, it is not clear what sort of programmatic role they play in the feature composition process.

   Traceability

The technique of virtual annotations has been proposed to support traceability using views [Kästner 2010] which we identify as the third property of traceability, namely traceability between SPL and product code as views enable separating out code of a variant from that of SPL. Other properties are not considered.

2. **Features as Units of Modularity in a Given Modularity Mechanism** In this category of feature implementations, features are represented using other modularity mechanisms. Any modularity representation that is capable of composing program deltas can be used to implement features (i.e., they can support the notion of features as optional and/or incremental unit of functionality). Examples of this category of techniques include implementing features in AspectJ via aspects [Kästner et al. 2007], in ObjectTeams with teams and roles [Hundt et al. 2007; Lopez-Herrejon et al. 2005a], in Jiazzi via atoms and units, in Scala via traits, in Hyper/J via hyper-slices [Lopez-Herrejon et al. 2005a] and in CaesarJ with aspects and roles [Mezini and Ostermann 2004].

    Feature Representation

    While concrete features are represented in terms of various modularity representations such as aspects, teams and roles, atoms and units, traits, and hyper-slices, none of them contain an explicit representation of modeling entities. The specification of composition has to be given in some format specific to the way feature are represented using the given modularity representation [Hundt et al. 2007; Kästner et al. 2007; Lopez-Herrejon et al. 2005a; Mezini and Ostermann 2004] and none of the techniques consists of a general mechanism of specifying modeling entities as it is possible with languages for feature modeling.

    Feature Composition

    When features are represented in terms of other modular entities, composition of such entities has to be modeled in such a way that it emulates the composition of concrete features they represent. Since the ability to add functionality (composed deltas together or with a base program) is the focus, the rest of the concern-specific functionality remains unused. Each modularity mechanism is in fact a representation of a concern, e.g., aspects represent concerns that signify crosscutting functionality [Apel 2007]. While aspects can aid in representing static and dynamic crosscutting features, in an SPL where such features are less in number compared to non-crosscutting features [Apel 2007; Mezini and Ostermann 2004], many of the capabilities of aspects remain unused [Kästner et al. 2007]. Similarly, modular entities called teams and roles in ObjectTeams represent the concept of roles at the programming language level [Herrmann 2007]. Traits in Scala are a modular representation of reusable behavioral concerns [Ducasse et al. 2006; Scharli et al. 2003], whereas atoms and units in Jiazzi are modular entities that capture the essence of Java packages with external linking and separate compilation

abilities [McDirmid et al. 2001; Xin et al. 2004]. We can observe that representing features is not the primary purpose of any of these modularity mechanisms. To represent concrete features, the representation and composition of each of these modular entities has to be adjusted which introduces a level of indirection as it were in both the representation and composition of concrete features.

The concrete features can also be implemented as hyperslices in Hyper/J [Lopez-Herrejon et al. 2005a]. Hyper/J represents concerns at a level of abstraction higher than that of individual concerns [Ossher and Tarr 2000b], or kinds of concerns. Its an implementation of multidimensional separation of concerns (MDSOC). A software system in Hyper/J is implemented with the idea that it can contain multiple concern dimensions and these concerns can be separated by projecting individual programs of the system by projecting their execution along concern hyperplanes [Ossher and Tarr 2000a;b]. Features can be implemented as hyperslices that group all units implementing the feature [Lopez-Herrejon et al. 2005a]. While the idea of higher abstraction in MDSOC is appealing, there are many drawbacks when the treatment of a individual concern such as features is concerned. For example, Chitchyan et al. [Chitchyan et al. 2003] found that in Hyper/J primary requirements, i.e., features, can not be traced in the composed system and it is difficult to comprehend how the composed system is affected when addressing additional requirements.

Traceability

None of the techniques above consider traceability explicitly, though traceability approaches have been suggested for AspectJ and Hyper/J [Clarke and Walker 2001].

3. **Features as Explicit Language Entities**

In this category of feature implementation techniques, concrete features are implemented as language entities themselves as compared to other two categories. Examples include the AHEAD tool suite [Batory 2004], FeatureC++ [Apel et al. 2005] and FeatureHouse [Apel et al. 2009b].

Feature Representation

All the techniques specified above, treat features as refinements to base classes. The feature-related code, i.e., the refinements, are stored in folders and the relations between features are emulated with a folder hierarchy. The fact that given program deltas belong to a feature is indicated using layer constructs added to Java. The conceptual features are not explicitly represented in AHEAD and FeatureHouse.

In addition to refinements stored in folder hierarchy, FeatureC++ also provides support for XML-based description of an SPL. It contains an external API that provides access to the feature model stored in XML format and enables instantiating the SPL (i.e., generate variant). It pro-

vides checks to ensure that an SPL instance to be generated is valid based on the feature model.

Feature Composition

The composition of concrete features represented as layers and class refinements is carried out using a textual description of features called equation files. The equation files are also used to specify inclusion and exclusion constraints. The equation file treats the folder hierarchy as a feature model to determine whether features specified indicate a valid variant. Given that a variant is valid, it is composed by adding refinement code to base classes. AHEAD furthermore enables checking safe composition of a given SPL, i.e., to determine whether all valid variants can be compiled safely, by converting relation between features to propositional formulas and applying satisfiability solvers to them [Thaker et al. 2007].

FeatureC++ extends the concept of static composition to include dynamic composition of features from a single code-base [Rosenmüller et al. 2008b; 2009]. This is achieved by transforming all refinements of a class (called refinement chain) to a delegation hierarchy using the decorator pattern. Since refinements must consider ordering, instantiating an SPL has to take into consideration the correct ordering of refinements before transforming the refinement chain to corresponding delegation hierarchy. Rosenmüller et al. have also presented an approach where an SPL contains multiple inter-dependent SPLs [Rosenmüller and Siegmund 2010; Rosenmüller et al. 2008a].

Traceability

Out of these techniques, only FeatureC++ achieves traceability between feature models and implementations at both SPL and product/variant level (i.e., traceability property (b) in Figure 3.3 on page 35) by using XML-based description of an SPL at run-time [Rosenmüller and Siegmund 2010; Rosenmüller et al. 2008a].

There is a progression in the feature implementation techniques from using preprocessors and annotations for representing concrete features, to using other modularity mechanisms for the same and representing them as language entities.

Based on this analysis, we conclude that using a language representation of concrete features makes them explicit in the code. Compared to preprocessor-based techniques, explicit language representation offers opportunities to support composition in a more streamlined manner. Between techniques that represent concrete features in terms of other modularity mechanisms and those that dedicate a specific representation such as refinement-based approaches, the latter offer less convoluted and more direct semantics for composition. But refinement-based approaches suffer problems some of which are representation-specific such as hierarchical features, and other problems that we deem to be

related to the lack of explicit representation of the rest of the feature implementation entities apart from concrete features. For instance, lack of traceability due to non-availability of explicitly represented SPLs and variants results in lack of reuse and lack of support for run-time composition.

While there have been attempts to make modeling entities first-class in the form of architectural variabilities [Loughran et al. 2008], similar attempts have not been made toward modularizing concrete features as first-class entities. All the implementation techniques, including those that represent features as language entities focus specifically on concrete features, the rest of the feature implementation entities as well as feature modeling entities are not explicitly represented. In the following, as the second step of our solution, we propose a language representation that integrates feature modeling and feature implementation entities in a language representation focused on raising their status in a host language.

Toward a New Representation

The analysis of representation and composition of feature domain entities reveals that different techniques opt to represent different feature modeling and implementation entities at various levels of abstraction depending on design choices in these techniques. Furthermore, in both feature modeling and implementation techniques, we notice that a language representation of various feature domain entities, either as an external DSL or as a language extension, offers explicit treatment of features with their own representation instead of an indirect handling of required semantics.

We refer to the research questions we formulated in Section 3.2.4 on page 36. Regarding the first question, we propose that following feature domain entities should be explicitly represented at the level of language:

- In feature modeling entities, feature models of an SPL and it variants as well as individual conceptual features should be explicitly represented.
- In the feature implementation entities, the exact counterparts of feature modeling entities specified above should be explicitly represented.

Regarding the second question, we propose that the representation of these entities be *first-class*. We explain what we mean by this in the next section.

3.4 Concept of First-class Features

In this section, we first elaborate the meaning of first-classness of entities in programming language in general.

3.4.1 Nature of First-class Language Entities

The *first-class*ness of certain entities in a given programming language indicates the degree to which such entities can be addressed in different situations using various expressions in the programming language and the extent to which they can be manipulated, in some cases at different times during the execution of a program written in this language. The term **first-class** was coined by Christopher Strachey in 1967 while describing the nature of procedures in ALGOL [Strachey 2000]. Procedures in ALGOL were deemed to be second-class because they can appear in another procedure as an operator or as an actual parameter but there are no other expressions involving procedures or whose result are procedures (that is they cannot be used as return values).

The term first-class has later been used by many researchers to indicate *increased level of manipulation* of certain representations in a programming language. In some cases, attempt is made to raise the status of the representation of specific concerns in the given programming language. Cointe showed that classes can be implemented as first-class entities using meta-classes, giving them a uniform and reflective definition [Cointe 1987].

Johnson and Duggan stated that continuations, entities that enable controlling the execution order of computational steps, have been used in Scheme in a variety of situations because of their first-class status and proposed to raise stores, a version of computer memory, to first-class status in the GL programming language so that they can be assigned as values of variables [Johnson and Duggan 1988]. Ernst presented the concept of object sets, a collection of objects of different classes, to be first-class entities in a language in which the emphasis is on *expressing the required primitives* such that object sets can be *modified during their lifetime* which would correspond to a change in a member class [Ernst 2008].

Other researchers similarly indicate that *giving an explicit and dedicated representation for some concern aimed at exploring all possibilities* related to that concern is tantamount to making it first-class in a given programming language, e.g., Shaw proposed to elevate component connectors, a representation of interactions and connections amongst components, to be first-class entities entitled to their own specification and abstractions [Shaw 1993].

We now combine the concept of first-classness of language entities with the idea of language representation of feature domain entities. Our analysis pointed to the fact that explicit and dedicated language representations are better than other representations. But the language representation we came across could be deemed to be passive in nature because not all required feature domain entities are represented in these language representations and they do not present the possibility of addressing feature domain entities in the program, increased level of manipulation, and modifications during the life time of the program. We believe that making the language representation of feature domain entities first-class would offer even better opportunities toward achieving unified

representation, composition, and traceability. We now extrapolate certain requirements targeted at making the suggested feature domain entities first-class.

3.4.2 Requirements of First-class Representation

While first-class status of entities in a programming language enable better usage and manipulation, when elevating features domain entities to first-class status, we also have to integrate other facets that we found to be crucial in our analysis toward merging feature modeling and feature implementation at the same time providing support for traceability. Below, we enlist all facets of such a representation as requirements to be fulfilled whenever features are to be first-class. It is the combined treatment of these requirements that enable features (and the rest of the feature domain entities) to be first-class.

1. ***First-classness in a Host Language*** Instead of using other mechanisms to represent and compose features, features should be represented as native first-class entities in the host language. This means that depending on the host language, feature domain entities should be represented at the same level of abstraction as the entities that have maximum degree of first-classness in the host language. A consequence of representation at such a level of abstraction may be that feature domain entities can be used not only as values that can be assigned to variables (discussed earlier as a characteristic of first-class entities) but also be used to represent instantiation.
2. ***Uniformity*** First-class features should represent conceptual and concrete features uniformly. More specifically, feature modeling and implementation entities should be represented by the same language entities. That means, a first-class representation of an SPL represents the feature model from the modeling perspective while at the same time it also represents the SPL at the level of implementation. A variant entity represents product/variant model at the modeling level and at the same time it also represents the variant from the feature implementation perspective. Using this variant entity, changes can be reflected to the product/variant model as well as the actual variant code at the same time. A feature entity represents a conceptual feature and its concrete counterpart simultaneously. The entities representing constraints become part of the SPL entity and are used toward deriving a valid variant entity.
3. ***Subsumed Checking*** Semantics of the first-class features should be rich enough to subsume safe composition of features. That means, as programming language entities of their own right, various aspect of type checking should be integrated as prerequisites toward safe composition of features to obtain variants. An SPL entity should be checked so that the underlying feature model is a valid feature model, e.g., it should not contain a feature with two parents. A variant entity should be checked with respect to its parent SPL (similar to decision models discussed earlier)

so that the variant entity contains exactly those features that are defined in the SPL entity and the features chosen indicate a valid decision based on relations and constraints. Similarly a feature entity is defined within the context of an SPL entity and the reference of this feature entity are checked to confirm that these uses refer to the original SPL entity.

4. ***Identity*** The identity of features should be retained throughout the life cycle of an application to enable a more controlled manipulation of the functionality of features. This requirements aims at the property of first-class entities that they are manipulable at different times during the execution of the program. This requirement also indicates that the identity of features that are composed to obtain a variant should not be lost post composition. The identity requirement can be fulfilled using meta-classes or metaprogramming based on what the host language supports more readily.
5. ***Extensibility*** When implementing the four requirements stated above, it is required, depending upon the host language, to extend its syntax and semantics. New keywords indicating various feature domain entities need to be added to host language syntax. Its semantics must be extended so that it supports typing of various entities and provides mechanism to retain the identity of various feature domain entities. This means that a given host language should posses efficient means of language extension. This can be also be helpful toward integrating more capabilities of features as they are discovered to their basic representation.

When feature domain entities are represented in a such a manner, we get what we refer to as **first-class features**. First-class features indicate a first-class representation of feature domain entities in a host language. In the next section, we relate the concept of first-class features to our earlier discussion of views on the mapping between problem and solution spaces.

3.4.3 Integrating the Views Using First-class Features

Earlier in Section 3.1.1 on page 28, we introduced the notions of conceptual and concrete features and explained the divide between them in terms of the dichotomy between problem space and solution space. Then, in Section 3.1.2 on page 28, we discussed the configuration view and the transformational view on the mapping between problem and solution spaces favored by feature modeling and feature implementation techniques respectively. We proposed and elaborated on another view on this mapping in Section 3.2.3 on page 34, called the traceability view, that requires both the configuration and transformation knowledge in order to establish traceability paths between various feature domain entities.

We build on these concepts and explain how the requirements of first-class features offer the opportunity to integrate these views. We illustrate this in

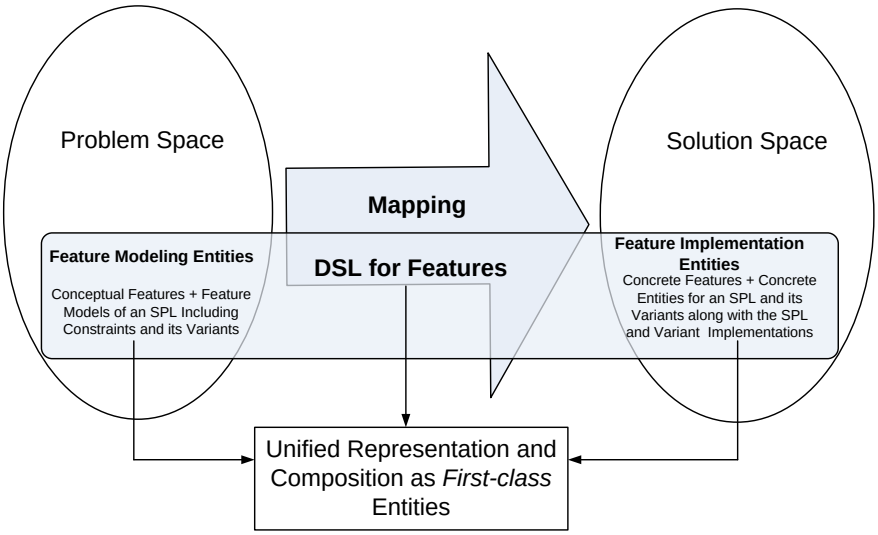

Figure 3.6: Integrating Feature Domain Entities with First-class Representation

Figure 3.6 which shows that the requirements of first-class features can be realized by integrating the representations of the concepts in problem space with implementation artifacts in solution space.

This can be achieved using an internal DSL atop a host language. Such a DSL would target domain-specific abstractions which in our case are the feature domain entities and implement them at the same level of abstraction as the entities with highest degree of first-classness in the host language. The implicit mapping between problem space and solution space fulfills the uniformity requirement as the same entity in this DSL represents both feature modeling and feature implementation counterparts. Domain-specific concrete syntax of our DSL would enable representing various domain entities, relationship between them, as well as operations on them using a syntax that closely mirrors that of the host language. Domain-specific checking in our DSL fulfills the subsumed checking requirement by integrating type checking of feature domain entities as type checking of DSL constructs.

We also propose to create an architecture around this DSL which is required to ensure proper instantiation of various feature domain entities. For example, when creating multiple variants using this DSL, we have to ensure that their coexistence does not conflict with the natural execution of program as it happens in the host language. This architecture would also take care of retaining the identity of various feature domain entities so that they can be addressed at different times during the execution of a program written in this DSL and they can be modified in valid ways. In the next section, we describe our choices of

host language to complete the second step toward the solution to achieving the unified perspective in FOSD.

3.4.4 Choice of Host Language

In the following, we describe which host language we choose to implement first-class features in terms of requirements obtained earlier.

Java

Java is prolifically used in most of the feature modeling techniques (via UML to Java code generation), as well as in the feature implementation techniques in FOSD. Instead of enlisting them, we point out the only exceptions which are COVAMOF [Sinnema et al. 2004b] implemented as Visual Studio plugins targeting .NET platform, Koalish [Asikainen et al. 2003] generating C code in feature modeling techniques, C-based #ifdef preprocessor [Spencer and Collyer 1992], FeatureC++ [Apel et al. 2005], and implementing features using traits in Scala [Ducasse et al. 2006; Scharli et al. 2003]. We therefore choose Java as one of the host languages to implement first-class features in. This would also give us the opportunity to corroborate treatment of feature domain entities in Java implemented as first-class features by comparing them with other Java-based feature modeling and implementation techniques.

Ruby

Ruby is a dynamic programming language [RubyHome; Stewart 2001; Thomas and Hunt 2000]. It has no compile-time and supports extensive metaprogramming and unconventional paradigms (compared to Java) such as open classes and modules that can be mixed-in (as opposed to super classes and interfaces in Java) [Flanagan and Matsumoto 2008]. To our knowledge, there are no feature implementation techniques in Ruby. Implementing first-class features in Ruby would give us insights into how the difference in the execution model and other capabilities affect the treatment of first-class features as opposed to their implementation in Java and in turn give us better understanding of core mechanisms in a host language that can be used to implement first-class features.

In Chapters 4 and 5, we elaborate our implementation of first-class features in Java and Ruby respectively as the third step toward the solution.

3.5 Summary

This chapter took a whirlwind tour of capabilities of features toward unearthing their dual nature as conceptual and concrete entities. We found that most FOSD techniques focus on feature modeling or feature implementation without considering traceability properties. An explicit representation of a set of feature domain entities at the level of language enables achieving a unified treatment of all feature domain entities with implicit traceability. Moreover, dedicating a representation that is focused at raising the status of feature domain entities within a host language is better than implementing them indirectly in terms of other modularity mechanisms. Such a representation would streamline composition as well. We extrapolated a set a requirements that would achieve such representation of feature domain entities which we call first-class features. We chose Java and Ruby as host languages to implement first-class features in. In the next two chapters, we elaborate our implementations of first-class features, first in Java called FeatureJ and then in Ruby called rbFeatures.

Chapter 4

FeatureJ

> [...] when you actually sit down and code something, you learn things that you didn't get from thinking about them in modeling terms [...] You realize that - oh - there are other abstractions I can use that you only get from working with them.
>
> MARTIN FOWLER
> *in Agile Software Development Ecosystems*

In this chapter we elaborate how the requirements of first-class features are implemented in Java using the JastAdd extensible compiler system to obtain FeatureJ. We first describe JastAdd's extensibility mechanisms followed by implementation details of FeatureJ including its syntax and semantics for feature domain entities and its overall architecture.[1]

4.1 JastAdd- An Extensible Java Compiler

In this section, we elaborate the mechanisms available in JastAdd toward implementing FeatureJ. Specifically, we explain JastAdd from the point of view of how the domain-specific abstractions in FeatureJ can be implemented, how JastAdd enables domain-specific concrete syntax for the same, and the ways in which domain-specific checking is made possible in JastAdd.

[1] This chapter shares material with the MCGPLE'08 paper *'Features as First-class Entities - Toward a Better Representation of Features'* [Sunkle et al. 2008b], the technical report *'Representing and Composing First-class Features with FeatureJ'* [Sunkle et al. 2009] and the RAM-SE'10 paper *'Using Reified Contextual Information for Safe Run-time Adaptation of Software Product Lines'* [Sunkle and Pukall 2010].

JastAdd is an aspect-oriented extensible compiler construction system [Ekman and Hedin 2007b;c; Hedin and Magnusson 2003]. It is based on the idea that the abstract syntax tree (AST) of a program can be used as the main data structure when implementing standard compiler analyses and adding new analyses, hence the name **J**ava **ast** **Add**ition. It supports typed access methods for traversing the AST [Ekman and Hedin 2004b]. It enables the use of both declarative and imperative programming techniques in implementing the compiler [Ekman 2004]. The declarative specification of complier analyses is achieved using Rewritable Reference Attribute Grammars (ReRAGs). The imperative programming is allowed using ordinary Java code which is woven into AST nodes using aspects [Ekman and Hedin 2004a].

Over the period, concepts in JastAdd have been extended to include circular ReRAGs that are used to implement a variety of metrics [Magnusson and Hedin 2007], declarative flow analysis which is used to implement innovative compiler analyses such as dead assignment finding [Nilsson-Nyman et al. 2008], and refactoring extension [Schäfer et al. 2009] which is used to implement various refactorings to Java source code, etc. In the following, we elaborate precisely those capabilities of JastAdd, which we use in implementing FeatureJ.

4.1.1 Object-oriented AST and Aspects

The most important characteristic of JastAdd is its use of the AST as the main data structure. Any Java extension implemented using JastAdd contains following parts: an abstract grammar that defines the structure of the AST, behavior specifications that define the behavior of the AST nodes, a context-free grammar that defines how text is parsed into ASTs, and an input program written using the extension that reads the input file, runs a parser to build the AST, compile it either by generating byte-code or transforming the AST to Java files and compiling them [Ekman and Hedin 2007c]. This is illustrated in Figure 4.1.

Nearly everything in JastAdd revolves around the AST, both figuratively and literally. JastAdd treats the nodes in the AST as instances of Java classes arranged in a subtype hierarchy [Ekman 2004]. We explain this with the help of specifications of while statement in JastAdd.

Listing 4.1: JastAdd Abstract Grammar for `while` Statement

```
1 abstract Stmt;
2 abstract BranchTargetStmt : Stmt;
3 WhileStmt : BranchTargetStmt ::= Condition:Expr Stmt;
```

An AST class represents a nonterminal, a production, or a combination of these two. JastAdd creates the subtype relations between the classes of the AST depending on the left hand side of the production whereas subtree nodes

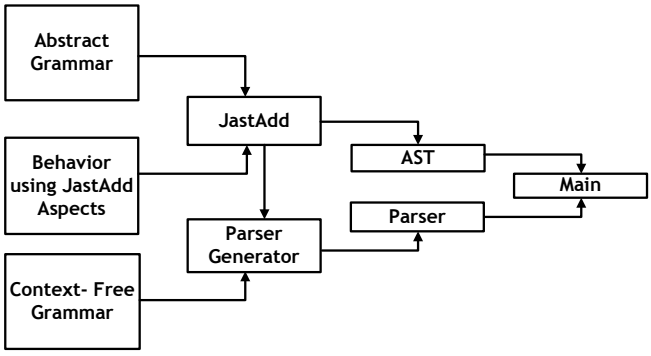

Figure 4.1: JastAdd Compiler Generation Architecture [Ekman and Hedin 2007c]

are generated using the right hand side. Starting with any node specified as `abstract` in the abstract grammar, all AST nodes inherit from the type ASTNode. Listing 4.1 results in the type hierarchy shown in Figure 4.2. This enables generic AST traversals [Ekman 2004] using the generic getParent() and getChild(int index) methods.

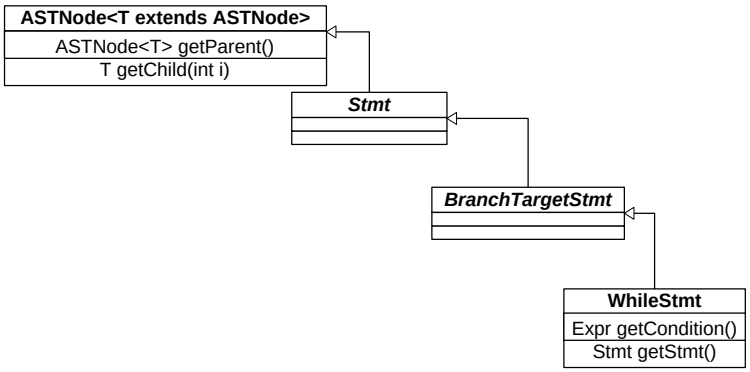

Figure 4.2: Type Hierarchy for `while` Statement

JastAdd is independent of the parser used to generate the AST [Ekman and Hedin 2004b]. Any parser generator can be used to create a parser so long as it generates the same AST as expressed in the abstract grammar. For example, CUP and JavaCC parser generators have been used to generate parsers with an LALR parser grammar and LL(k) parser grammar respectively in JastAdd [Ekman and Hedin 2004b]. JastAdd uses the Beaver parser generator (an LALR-based parser generator) to generate parsers for both Java 1.4 and Java 1.5 compilers as well to represent other extensions. Listing 4.2 shows the

parser grammar for the *while* statement which is used along with the abstract grammar specified in Listing 4.1 to obtain AST node classes as shown in Figure 4.2.

Listing 4.2: Beaver Parser Generator Grammar for `while` Statement

```
1 Stmt statement = while_statement.w  {: return w; :} ;
2 WhileStmt while_statement =
3  WHILE LPAREN expression.e RPAREN statement.s
4  {: return new WhileStmt(e, s); :} ;
```

JastAdd supports static aspect-oriented specifications [Ekman and Hedin 2004a] along with imperative Java code. All the behavioral additions including the compiler analysis-specific code such as name and type analysis, error checking, etc., is put together in JastAdd's own aspect mechanism which is based on AspectJ's introduction and inter-type declaration mechanisms [Kiczales et al. 2001].

4.1.2 Rewritable Reference Attribute Grammars

In the following we first review JastAdd's mechanism of RAGs followed by ReRAGs which extend RAGs with a conditional rewrite system.

Reference Attribute Grammars

The reference attribute grammars (RAGs) is an object-oriented extension to Knuth-style attribute grammars [Knuth 1990]. Attribute grammars are characterized by the fact that each node in the AST has an attribute. The relations between different attributes are defined using equations. The attribute evaluator in JastAdd computes the attribute values so that the AST becomes consistently attributed. This is checked statically and any missing attributes are requested to be defined. The attributes are defined as methods of classes of AST nodes. An attribute is accessed by calling the method it represents [Ekman and Hedin 2004a]. Such attribute evaluation may result in non-trivial traversals of the AST where the same node is accessed many times. Therefore, JastAdd caches the computed value of an attribute so that the same attribute is not computed twice. JastAdd's uses RAGs ubiquitously in its implementation of the Java compilers. For instance, whenever an access of an entity is to be connected to its declaration it is done so using RAGs. There are two types of attributes used in JastAdd, synthesized and inherited attributes as discussed next.

Synthesized Attributes

A synthesized attribute of an AST node resides in the node itself. These attributes can be accessed from a node's ancestors. This way, the information can be propagated upwards e.g., type information from an operand to its enclosing expression [Ekman 2004]. The keyword `syn` denotes a synthesized attributes and its equation is specified using the keyword `eq` as shown in Listing 4.3. The keyword `lazy` indicates that value of this attribute is cached by JastAdd [Ekman and Hedin 2007b].

Listing 4.3: A Synthesized Attribute in JastAdd

```
1  aspect UnreachableStatements {
2    syn lazy boolean Stmt.canCompleteNormally() = true;
3    eq WhileStmt.canCompleteNormally() =
4     reachable() &&
5      (
6      !getCondition().isConstant() ||
7      !getCondition().isTrue()
8      ) ||
9     reachableBreak();
10    ...
11 }
```

Synthesized attributes act like virtual methods [Ekman and Hedin 2007c]. Listing 4.3 shows the definition of normal completeness of `while` statement (see Java Language Specification (JLS), Section 14.20 Unreachable Statements [Gosling et al. 2005]). The canCompleteNormally() attribute (which is implemented as a method) returns true by default for the Stmt class whereas for WhileStmt class, a definition is provided according to the JLS.

Inherited Attributes

The inherited attributes of an AST node reside in an ancestor of the node. Such inherited attribute of node A defines the behavior of the corresponding declaration of the same attribute in the subtree where a node of the same type as A is the root [Ekman 2004]. The result of this is that inherited attribute enable child nodes to obtain information about their context (position in the AST, surrounding AST node, hosting type, etc.)

Listing 4.4 shows the definition of the reachable attribute for WhileStmt which was used in Listing 4.3. The keyword `inh` indicates an inherited attribute and the equation defines the reachable() attribute for each statement in the block of a `while` statement. Such combined usage is evident in many places in JastAdd where synthesized and inherited attributes are employed together as shown in Listings 4.3 and 4.4 to move information around in the AST.

Listing 4.4: An Inherited Attribute in JastAdd

```
aspect UnreachableStatements {
 inh boolean Stmt.reachable();
 eq WhileStmt.getStmt().reachable() = reachable()
  && !getCondition().isFalse();
}
```

Context Dependent Rewrites

ReRAGs extend RAGs with the ability to rewrite certain nodes automatically and transparently. Instead of the demand driven evaluation of attributes in RAGs, which consists of evaluating an attribute when its value is read, ReRAGs extends the basic evaluation scheme by enabling rewriting AST during attribute evaluation.

The rewriting of a node is triggered by the first access of that node. Internally, getting to any node (also for the first visit) takes place using the getChild(int index) method, which is also used to trigger rewrites. The rewrite rules for each node type are translated into a rewireTo() method that checks rewrite conditions and returns the rewritten tree. A rewriteTo() method of a node is called iteratively until all rewrite conditions are false. This results in a rewritten subtree where the original node was the root [Ekman and Hedin 2004a]. Any latter accesses of the node view the rewritten node rather than the original. When using ReRAGs, it is possible that a rewrite that alters the tree structure affects an already computed attribute value, JastAdd caches only those attributes that cannot be affected by later rewrites.

Listing 4.5: AST Rewrite Specification in JastAdd

```
rewrite N {
 when ( condition )
  to
  R {
   ...
   return expression;
 }
}
```

A rewrite rule is specified using a condition that states when the rewrite is applicable and the resulting tree. The rewrite rule application order and the related tree traversals are defined based on relations between attributes [Ekman 2004]. ReRAGs therefore form a conditional rewrite system where conditions and rules can utilize the contextual information in the AST using the attributes [Ekman and Hedin 2004a]. E.g., in Listing 4.5, it is indicated that a node of type

N is to be replaced by a subtree with the root of the type R when `condition` is true. JastAdd ensures that the rewrites are type consistent, which means that regardless of the context of the node, the replacement of N by R results in a type consistent AST with subtype relation preserved [Ekman and Hedin 2004a].

ReRAGs simplify compiler implementation in many ways such as *semantic specialization*, e.g., differentiating between the qualified names during name analysis [Schäfer et al. 2008], *making implicit behavior explicit*, e.g., implicit constructors of classes in Java such that a class without a constructor is rewritten at the level of AST to a class with an implicit constructor, and *replacing concrete syntax with the intended semantics*, e.g., overloaded binary addition operator for strings is replaced by call to the concat() method [Ekman and Hedin 2004a].

4.1.3 Name and Type Analysis in JastAdd

In this section, we review how name and type analysis for Java is carried out in JastAdd using the AST is the main data structure.

Name analysis

The name analysis of Java source means binding each access of an entity through its identifier to the declaration of the entity [Ekman and Hedin 2007c]. The name analysis aspects in JastAdd are implemented in Lookup*.jrag files where * indicates separate .jrag files (where jrag extension indicates RAG-based attributed specification in JastAdd) for name analysis of types, variables, methods, and constructors. The problem of name analysis is divided into two subproblems:

Visibility and Scopes of Entities The visibility rules in Java must handle various scopes as well as their combinations such as inheritance and nested scope and qualified access to remotely defined entities. These rules are implemented using a combination of inherited and synthesized attributes. While the inherited attributes lookup* define the scope of an entity, the synthesized attributes return entities defined in that scope. This is illustrated in Listing 4.6. Lines 2 and 3 in Listing 4.6 declare the inherited attribute lookupVariable for Stmt and Block classes. Lines 4-10 define this attribute for Block class which calls a method localVariableDeclaration(String name). This method is defined as a synthesized attribute in Lines 11-17 and returns the set of local variable declarations in a Block with the given name of a local variable. Lines 18 and 20 declare and define method declaresVariable() which returns true if a VariableDeclaration has the same name as the given name which is being looked-up.

Recall that the block of `while` statment is obtained through getStmt() method and local variables can be looked up by calling lookupVariable() method of that block.

Listing 4.6: Name analysis of Local Variables in `while` Statement

```
aspect VariableScope {
 inh SimpleSet Stmt.lookupVariable(String name);
 inh lazy SimpleSet Block.lookupVariable(String name);
 eq Block.getStmt(int index).
  lookupVariable(String name) {
    VariableDeclaration v = localVariableDeclaration(↩
        name);
    if(v != null && declaredBeforeUse(v, index))
      return v;
    return lookupVariable(name);
  }
 syn lazy VariableDeclaration
  Block.localVariableDeclaration(String name) {
    for(int i = 0; i < getNumStmt(); i++)
      if(getStmt(i).declaresVariable(name))
        return (VariableDeclaration)getStmt(i);
    return null;
  }
 syn boolean Stmt.declaresVariable(String name)
  = false;
 eq VariableDeclaration.declaresVariable(String name)
  = name().equals(name);
}
```

Meaning of Names As seen earlier, JastAdd uses a parser generator to generate a parser which is used along with the AST generated by JastAdd itself. While a parser generator builds general Access nodes (each denoting an access of a named entity) for all named entities, JLS states that these should be classified based on their context (see JLS [Gosling et al. 2005], Section 6.5). This is treated as a case of semantic specialization as stated earlier, for which context-dependent rewrites are used for differentiating between names based on their context, i.e., where the nodes that define the named entities are situated in the AST [Schäfer et al. 2008].

Type analysis

The type analysis of Java source means to resolve types of various entities including conversions between typed entities and verify their correctness according to the JLS [Ekman and Hedin 2007c]. The type analysis aspects in JastAdd are implemented in TypeAnalysis.jrag, TypeHierarchyCheck.jrag, and TypeCheck.jrag files. The problem of type analysis in JastAdd is divided into two subproblems:

Computing the Type of Expression Java contains explicitly and implicitly declared types. Classes and interfaces are examples of explicitly declared types for which, the declaration nodes are treated as type representations. Primitive types and arrays are examples of implicitly declared types for which, JastAdd adds AST nodes as part of the attribute evaluation [Ekman and Hedin 2007c].

Determining the Subtype Relation The subtype relation is implemented in JastAdd by looking up the direct supertypes transitively. As all types are represented as AST nodes, such lookup is implemented as a traversal of AST using reference attributes that bind type declaration to their direct supertypes [Ekman and Hedin 2004a].

At the core of both the name and type analysis in JastAdd is the basic design principle of using the AST as the only data structure. Using the AST as the only data structure requires that types are represented by the nodes in the AST and the type of an expression is represented by a reference attribute that refers to the appropriate type node. Similarly, names are resolved based on looking up subtrees of the nodes in the AST that introduce a scope such as a `while` statement.

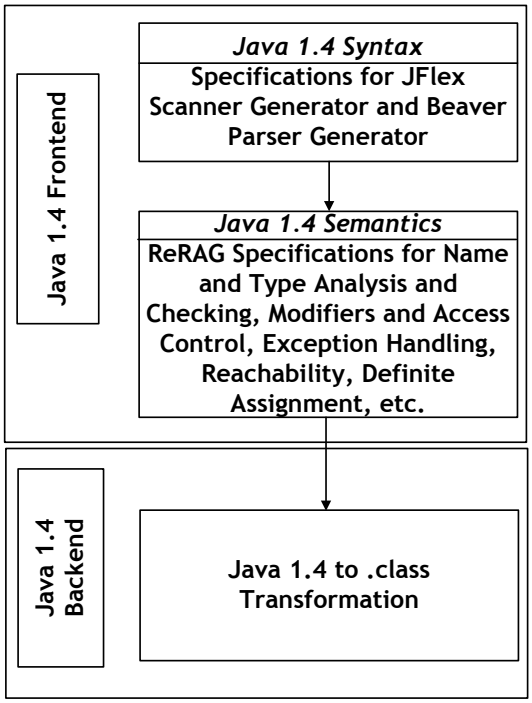

Figure 4.3: Java 1.4 Compiler using JastAdd

4.1.4 JastAdd Implementation of Java Compilers

In this section, we review how JastAdd puts together abstract and parser grammars, the AST and the parser, and all the compiler analyses to implement Java 1.4. Following this, we review how JastAdd demonstrates its extension capability by extending the Java 1.4 compiler to Java 1.5 compiler.

Java 1.4 Compiler

The Java 1.4 compiler in JastAdd is implemented as a combination of a frontend and a backend as shown in Figure 4.3 on the previous page. The arrows with filled triangles indicate that semantic component uses the syntactic component and later semantic component is used by byte-code generation. A set of Java files are input to the frontend which performs lexing, parsing, name and type analyses as well as other analyses and error checking, while the responsibility of the backend is to transform the AST into a set of .class files.

Scanner and parsers for Java 1.4 source are obtained using the JFlex scanner generator and Beaver parser generator respectively. As described earlier, while Beaver parser generator uses the JFlex scanner generator, JastAdd interfaces with the Beaver parser generator to build an AST from Java 1.4 source using the abstract grammar and the generated parser. Any syntactic errors are caught and reported during parsing itself. When there are no syntactic errors, various analyses are performed on the generated AST.

Listing 4.7: Error Checking in Java 1.4 Frontend of JastAdd

```
1  aspect ErrorCheck {
2   public void ASTNode.collectErrors() {
3     nameCheck();
4     typeCheck();
5     accessControl();
6     exceptionHandling();
7     checkUnreachableStmt();
8     definiteAssignment();
9     checkModifiers();
10    for(int i = 0; i < getNumChild(); i++) {
11       getChild(i).collectErrors();
12     }
13   }
14 }
```

The basic entry point to all compiler analyses in the Java 1.4 compiler implementation of JastAdd is the ErrorCheck aspect shown in Listing 4.7. The entire AST is traversed using collectErrors() method. Depending upon a Java language entity and its corresponding specification in the JLS, the AST node that represents this entity provides an implementation for zero or more of methods in the collectErrors() method.

Listing 4.8: Name Checking Class Instance Creation Expressions

```
aspect NameCheck {
 public void ClassInstanceExpr.nameCheck() {
   super.nameCheck();
   if(decls().isEmpty())
     error("can not instantiate " + type().typeName() +
      " no matching constructor found in " +
       type().typeName());
   else if(decls().size() > 1 && validArgs()) {
     error("several most specific constructors found");
     for(Iterator iter = decls().iterator();
        iter.hasNext(); ) {
        error("         " +
         ((ConstructorDecl)iter.next()).signature());
    }
   }
  }
}
```

JastAdd contains a ReRAGs specification of each of the methods called in the collectErrors() method. There are .jrag files NameCheck.jrag, TypeCheck.jrag, AccessControl.jrag, ExceptionHandling.jrag, UnreachableStatements.jrag, DefiniteAssignment.jrag, and Modifiers.jrag in the Java 1.4 compiler implementation in JastAdd that contain one or more related aspects. These methods in turn invoke name analysis, type analysis, and other required functionality defined in the rest of aspect in other .jrag files.

Listing 4.9: Checking Access Control in Class Instance Creation Expressions

```
aspect AccessControl {
 public void ClassInstanceExpr.accessControl() {
  super.accessControl();
  if(type().isAbstract())
    error("Can not instantiate abstract class "
     + type().fullName());
  if(!decl().accessibleFrom(hostType()))
    error("constructor " + decl().signature()
     + " is not accessible");
  }
}
```

As an example, consider the AST node ClassInstanceExpr which represents class instance creation expressions in Java using the `new` keyword. The ClassInstanceExpr class contains nameCheck(), typeCheck(), and accessControl() methods that mirror specifications in the JLS related to class instance creation expressions (see JLS [Gosling et al. 2005], Section 15.9). We explain each of the nameCheck(), accessControl(), and typeCheck() methods of ClassInstanceExpr in the following. Note that similar mechanisms are use in error checking nodes

of the entire AST, where attribute values for specific nodes are computed once when called first and then for subsequent queries if any, the cached values are returned.

Listing 4.8 shows nameCheck() method of ClassInstanceExpr in the aspect NameCheck defined in the NameCheck.jrag file. The attribute type() in Line 5 uses type lookup mechanism to find the node of TypeDecl or its subclasses using the access in the class instance expression, e.g, given `new T();` T is the access, the declaration of which is returned by the type() attribute. The attribute decls() in Line 4 on the other hand, returns the declarations of constructor T() for type T.

Lines 4-15 in Listing 4.8 check that (a) there is a constructor matching constructor T(), and if not, instantiation is not allowed and (b) there is exactly one constructor in T that matches the signature of T() in the expression `new T();`.

The accessControl() method ensures that the `new` keyword is not used with respect to an abstract class constructor (Lines 4-6) and that the expression `new T();` is not using a constructor that is not accessible from the current type (Lines 7-9) where the expression `new T();` is situated as shown in Listing 4.9.

Listing 4.10: Type Checking Class Instance Creation Expressions

```
1  aspect TypeCheck {
2   public void ClassInstanceExpr.typeCheck() {
3    if(isQualified() && qualifier().isTypeAccess()
4     && !qualifier().type().isUnknown())
5      error("*** The expression in a qualified
6       class instance expr must not be a type name");
7     if(isQualified() && !type().isInnerClass() &&
8        !((ClassDecl)type()).superclass().isInnerClass()
9        && !type().isUnknown()) {
10     error("*** Qualified class instance creation can only
11         instantiate inner classes and their
12          anonymous subclasses");
13    }
14    if(!type().isClassDecl()) {
15     error(" Can only instantiate classes, which "
16           + type().typeName() + " is not");
17    }
18     typeCheckEnclosingInstance();
19     typeCheckAnonymousSuperclassEnclosingInstance();
20   }
21 }
```

The typeCheck() method of ClassInstanceExpr in Listing 4.10 ensures that (a) if the class instance creation expression is qualified, then it must refer to inner member classes or their anonymous subclasses, (b) if it not qualified, then it must refer to a class (for both (a) and (b), see JLS [Gosling et al. 2005], Section 15.9.1). Furthermore, the typeCheckEnclosingInstance() method (Line

18) checks the compilation errors related to determining the correct enclosing instances and the typeCheckAnonymousSuperclassEnclosingInstance() method (Line 19) checks compilation errors related to the class instance expression occurring in a static context (see JLS [Gosling et al. 2005], Section 15.9.2).

According to the JLS, the semantics of class instance creation expressions need not directly consider exception handling, reachability conditions, definite assignments, and modifier checking which means that exceptionHandling(), checkUnreachableStmt(), definiteAssignment(), checkModifiers() are not defined in ClassInstanceExpr. All AST nodes similarly implement specific methods for checking a Java program for compilation errors. The source for the implementation of Java 1.4 and 1.5 compilers in Java actually contain comments specifying which JLS specification is being implemented. Using declarative form of JLS specifications, JastAdd maintains a very high compliance to the JLS [Ekman and Hedin 2007c]. Note that the order of method calls in the aspect ErrorCheck in Listing 4.7 is not important as the attribute evaluation is demand-driven as well as cache supported [Ekman and Hedin 2007b;c].

Extra libraries can be specified as .jar files to the frontend. It contains the functionality to load classes from .jar files which are used in various analyses. When there are no semantic errors, the AST can be input to the backend. The backend is implemented as an extension of the frontend. It extends the frontend and contains aspects for transforming each AST node to its byte-code equivalent which is written to a .class file. These .class files can be interpreted with the java.exe interpreter.

Java 1.5 Compiler

The frontend and backend in Java 1.5 compiler in JastAdd are implemented as extensions to Java 1.4 frontend and backend. The Java 1.5 implementation in JastAdd is shown in Figure 4.4. The arrows with filled triangles indicate how each component is used. Java 1.5 extensions such as enums, enhanced for statement, autoboxing, varargs, static imports, and generics with wild cards are implemented by extending pre-existing AST nodes, or adding new nodes to AST when required. Listing 4.11 is an example of how existing analyses are extended using aspect refinements.

The JLS requires that enum types can not be instantiated using the `new` keyword (see JLS [Gosling et al. 2005], Section 8.9). The expressions containing the `new` keyword are represented in JastAdd by the class ClassInstanceExpr. In order to enforce additional constraint on the enum types, the nameCheck method of ClassInstanceExpr in the aspect NameCheck is refined as shown in Listing 4.11.

The Java compiler javac in J2SE version 5.0 (implementation of Java 1.5) internally generates a class for an enum type. JastAdd's implementation of enums is represented by EnumDecl which extends ClassDecl. The fact that

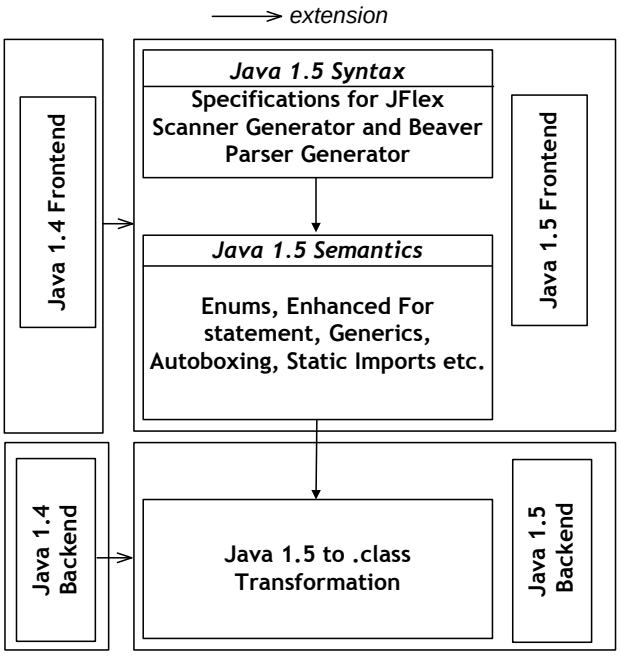

Figure 4.4: Java 1.5 Compiler using JastAdd

internally a class is generated based on an enum type is implemented using rewrites in Java 1.5 compiler implementation in JastAdd as shown in Listing 4.12.

Listing 4.11: Extending Name Checking to Accommodate Enum Types

```
1  aspect Enums {
2   syn boolean TypeDecl.isEnumDecl() = false;
3   eq EnumDecl.isEnumDecl() = true;
4   syn boolean BodyDecl.isEnumConstant() = false;
5   eq EnumConstant.isEnumConstant() = true;
6   refine NameCheck public void
7    ClassInstanceExpr.nameCheck() {
8      if(getAccess().type().isEnumDecl() &&
9          !enclosingBodyDecl().isEnumConstant())
10       error(``enum types may not be instantiated explicitly↩
             '');
11     else
12       NameCheck.ClassInstanceExpr.nameCheck();
13  }
14 }
```

Lines 5-32 in Listing 4.12 indicate that when no constructor has been specified

for an enum type, a default private and synthetic constructor with two parameters including p0 of String type and p1 of int type is generated (indicating the name and the value of the enum constant) with a call to superclass constructor super(p0,p1). This class is used by Java 1.5 backend to emit proper byte-code.

Listing 4.12: Using Rewrites to Generate Internal Classes for Enum Types

```
aspect Enums {
 rewrite EnumDecl {
   when(!done())
     to EnumDecl {
 if(noConstructor()) {
   List parameterList = new List();
   parameterList.add(
     new ParameterDeclaration(new TypeAccess("java.lang", "String"), "p0"));
   parameterList.add(
     new ParameterDeclaration(new TypeAccess("int"), "p1"));
   addBodyDecl(
     new ConstructorDecl(
       new Modifiers(new List().add(new Modifier("private")).add(new Modifier("synthetic"))),
       name(),
       parameterList,
       new List(),
       new Opt(
         new ExprStmt(
           new SuperConstructorAccess(
             "super",
             new List().add(
               new VarAccess("p0")
             ).add(
               new VarAccess("p1")
             )
           )
         )
       ),
       new Block(new List())
     )
   );
 }
 else {
   transformEnumConstructors();
 }
 addValues();
 return this;
 }
}
```

Listings 4.11 and 4.12 show that the enum type extension is implemented by (a) adding an AST node EnumDecl (which is added using the abstract grammar of Java 1.5), (b) refining pre-existing analyses as required, and (c) using rewrites to generate internal classes if required. Other Java 1.5 specific extensions are similarly implemented in JastAdd Java 1.5 compiler using a combination of these techniques.

4.1.5 Extending JastAdd's Implementation of Java

In this section, we elaborate the standard ways of adding language features to Java using JastAdd.

Our objective is to point out specific manner in which new functionality pertaining to the extensions is accommodated in JastAdd and which are also useful in implementing feature domain entities as required in FeatureJ. We enlist the same in the following:

Extending Java Types Whenever functionality of types in Java is to be extended or new kinds of types are to be added to Java, they can be represented as subclasses of TypeDecl or its subclasses in the JastAdd AST. For instance, enum types and generic types [Ekman and Hedin 2007c], non-null types [Ekman and Hedin 2007a] are all represented as subclasses of TypeDecl and its subclasses.

Another candidate for extending TypeDecl and its subclasses is domain-specific language extensions to Java [Ekman and Hedin 2004b]. For instance, in order to create a DSL for Matrix computation, matrices can be represented by a type in JastAdd by extending TypeDecl or its subclasses. The Matrix type would be a built-in type in such case and can reuse functionality and existing ways of implementing typed extensions to Java using JastAdd as described in the preceding section. This corresponds to the concept of implementing domain-specific abstractions as language entities [Czarnecki 2004].

Extending Name and Type Analysis and Error Checking Whenever an extension demands customized error checking, it can be implemented atop JastAdd Java compilers such that existing error checking analyses are refined and new analyses can be added as shown earlier in case of enum types (cf. Section 4.1.4 on page 67). Boustani and Hage have extended the Java 1.5 analyses in JastAdd to include extended type inference for better error diagnostic and error messages for generic method invocations [Boustani and Hage 2009]. Unlike other standard Java compiler implementations, JastAdd's analyses are modular and extensible [Ekman and Hedin 2007b]. Error checking can be implemented on per node basis and whenever required, can be refined. This ability is also being utilized for run-time assertion checking for modeling annotations in Java Modeling Language (JML) [Haddad and Leavens 2008]. Similarly,

support for multiple inheritance with new language constructs such as *requires* is provided [Malayeri 2008] in CZ language. The core implementation is based on extended name and type analysis and error checking to accommodate multiple inheritance by extending JastAdd's analyses (specified in the presentation slides accompanying the paper).
In case of DSLs, domain-specific checking [Czarnecki 2004] can be implemented as an extension of name and type analysis and error checking of the host language. The interaction of domain-specific entities represented as types in JastAdd with regular Java types can be safeguarded by extending JastAdd's analyses.

AST Transformation of the Extension Source to Java Whenever it is required to generate new classes from the existing Java source either for internal compiler usage (such as in the case of enum types shown earlier in Listing 4.12 on page 69), or explicitly, it is desirable to use JastAdd's conditional rewrite system. For instance, ContextJ which a context-oriented programming extension to Java uses JastAdd's AST transformation capabilities using rewrites to generate classes for its *layer* constructs [Appeltauer et al. 2010]. Similarly, if the extension demands additional classes to be generated when the original AST is processed, JastAdd's rewrite system can be used for this purpose.

In JastAdd, a new language feature is built based on existing AST hierarchy. All of the Java 1.5 extensions are implemented with subclasses of existing AST nodes [Ekman and Hedin 2007b;c]. Other Java-specific extensions are similarly implemented by JastAdd developers [Ekman and Hedin 2007a; Nilsson-Nyman et al. 2008] and other researchers [Boustani and Hage 2009; Malayeri 2008]. Combined with the implementations of non-standard language features [Åkesson et al. 2010; Appeltauer et al. 2010; Avgustinov et al. 2008; Haddad and Leavens 2008; Hedin et al. 2010] atop JastAdd Java 1.4 and 1.5 compilers, the JastAdd-based Java extensions consistently exhibit the same pattern of extensibility as discussed above.

4.1.6 JastAdd as the Compiler Construction System of Choice

As seen in the preceding sections, JastAdd provides a variety of concepts that are quite helpful in representing extensions to Java 1.4 and 1.5 compilers. While the initial learning curve may be steep due to the sheer number of the concepts involved, a developer can soon get comfortable with JastAdd as particular facets of extension can be implemented in specific ways. In the following, we elaborate why we preferred to use JastAdd instead of other extensible compiler construction systems:

Extensions at Both Syntactic and Semantic Levels While IDE-specific compiler implementations such as Eclipse Java Development Tools (JDT)

may provide suitable APIs for additional compiler analyses, they are less suited for language extensions that affect all parts of the host language. Source to source translators for Java similarly provide support only for syntactic extensions and do not support semantic extensions e.g., MetaBorg [Bravenboer and Visser 2004], Java Syntactic Extender [Bachrach and Playford 2001], and Jakarta Tool Suite [Batory et al. 1998]. The problem with these tools is the lack of context-sensitive translation capability [Ekman and Hedin 2007c]. Furthermore, error checking is performed on the translated code (from the extended source to Java source) which results in alignment mismatch between errors in the extended source and the translated source. Other approaches provide support on the semantic level but not at the syntactic level, e.g., OpenJava [Tatsubori et al. 2000] which uses a class-based macro system to implement a meta-object protocol in which semantic analyses are extended using metaobjects. Although this idea is quite novel in enabling manipulation of Java entities at various times during execution, it does not provide support for syntactic extensions. JastAdd on the other hand, enables extremely modular and intuitive syntactic extensions (separated into lexical and parsing components as shown in Figures 4.3 on page 63 and 4.4 on page 68) as well modular semantic extensions based on JastAdd aspects and ReRAGs.

Automated Scheduling of Compiler Analyses While there are extensible compiler construction systems that enable both syntactic and semantic extensions such as e.g., Polyglot [Nystrom et al. 2003] and JaCo [Zenger and Odersky 2001], they require manual scheduling of compiler analyses. For instance, while the AspectBench compiler for AspectJ was first implemented in Polyglot, it was re-implemented in JastAdd and the developers noted that the new implementation was half the size, twice as fast, modular (with better localization of concerns), supported composable extensions, and provided automated scheduling of complex compiler passes [Avgustinov et al. 2008].

Complete Implementation of Java 1.4 and Java 1.5 Compilers JastAdd is backed-up by compiler implementations of Java 1.4 and Java 1.5 which serve not only as an excellent source of examples but also prove the ease brought by novel idioms used in JastAdd. In case of large software such as extensible compiler construction system, it is often quite difficult to assess what happens where in the entire source code. How to achieve a particular task whether it is syntax or semantics related is difficult to fathom. With JastAdd's complete implementation of Java 1.4 and 1.5 compiler, one has to only accustom oneself with the basic concepts and see their implementation in the source code for Java 1.4 and 1.5 compilers. This makes the task of compiler extension quite easy and streamlined.

All the characteristics of JastAdd specified above come into play when implementing FeatureJ as we will explain in the next section.

Utilizing JastAdd Mechanisms for First-class Features

The focus of Section 4.1 was to investigate JastAdd's abilities and various mechanisms available in it toward implementing first-class features. We have elaborated the standard ways in which extensions and internal DSLs can be implemented in Java using JastAdd. We use JastAdd's way of representing extensions by extending types with extension to the AST and corresponding extensions at syntactic level to represent the domain-specific abstractions and their syntax in FeatureJ. Similarly, we use JastAdd's mechanisms of name and type analysis and error checking as well as its conditional rewrite system to implement domain-specific checking and overall semantics of FeatureJ. We elaborate how this is done starting with the next section.

4.2 FeatureJ Language Internals

FeatureJ is an internal DSL implementation of first-class features in Java, implemented using JastAdd's implementation of Java 1.4 and 1.5 compilers. FeatureJ uses domain-specific abstractions which are feature domain entities, creates a domain-specific concrete syntax for them, and provides domain-specific checking in terms of checking validity and correct compilation of variants. In the following sections, we elaborate each of these and then discuss the architecture that makes it possible for multiple variants as different versions of a Java program to co-exist and being acted upon at the same time.

4.2.1 Feature Domain Entities as JastAdd Types

As discussed earlier in Section 4.1, one of the ways in which JastAdd's extensibility features are utilized is by implementing new extensions as subclasses of TypeDecl node in the AST. Based on our discussion in the last chapter (cf. Section 3.4.3 on page 51), we need to represent feature domain entities in terms of semantically richest constructs in the host language. In Java these are types.

While it is controversial whether classes and the class type in Java are first-class, the Java Language Environment white paper specifies arrays to be first-class entities [Gosling and McGilton 1996]. More specifically, they are called first-class language objects, which have actual typed representation also available for run-time access implying that objects and classes are considered first-class in Java. Arrays are represented as subclasses of ClassDecl in the JastAdd representation, and as seen in Figure 4.5, ClassDecl is itself a descendent of the abstract class TypeDecl. We similarly implement the feature domain entities as descendants of the TypeDecl node, which in JastAdd parlance means that feature domain entities are implemented as types. Figure 4.5 illustrates the type hierarchy of various feature domain entities. This type hierarchy is expressed through the JastAdd abstract grammar specification as shown in Listing 4.13.

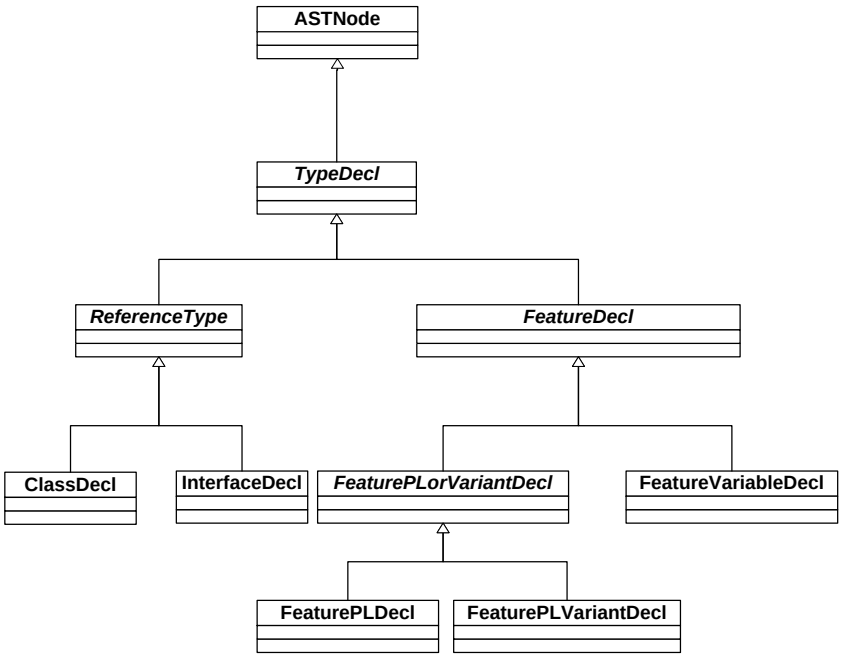

Figure 4.5: Type Hierarchy of Feature Domain Entities in FeatureJ

Listing 4.13: Abstract Grammar for Feature Domain Entities in FeatureJ

```
1  abstract FeatureDecl : TypeDecl ;
2  abstract FeaturePLorVariantDecl : FeatureDecl ;
3  FeaturePLDecl : FeaturePLorVariantDecl ::=
4     featurePLInnerExprs : FeatureExpr*
5     FPLCrOuterAccessExpr : FeatureExpr
6     [FPLConstraintExpr : FeatureExpr] ;
7  FeaturePLVariantDecl : FeaturePLorVariantDecl ::=
8     productLineUse : Access   <ID : String>
9     fVarSelectExpr : FeatureExpr;
10 FeatureVariableDecl : FeatureDecl ::= <ID : String> ;
```

Note that FeaturePLDecl and FeaturePLVariantDecl are subclasses of an abstract class FeaturePLorVariantDecl. We present an SPL and a product variant in this way because both represent feature models, the first representing the base feature model for an SPL and the second representing a feature model selection i.e., a product/variant model. The contents of a feature model and a product/variant model are represented through various feature expressions which are represented by the class FeatureExpr which itself subclasses the JastAdd abstract class Expr for all Java expressions.

The FeatureVariableDecl class represents a feature entity. In order to understand how we represent various code fragments that can be indicated to be part of one or more features, consider Figure 4.6, which shows various structures in a Java program that form a containment hierarchy.

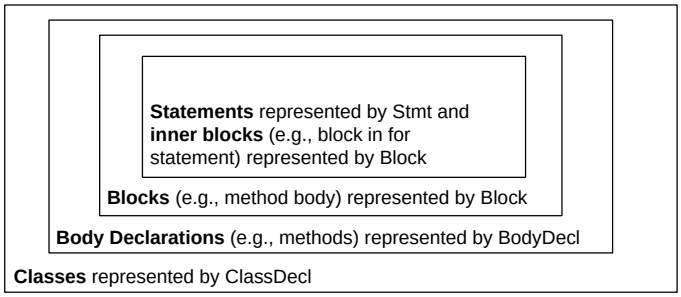

Figure 4.6: Containment Hierarchy of Language Constructs in Java

The body declarations in a class or interface that are part of a feature are represented by the class FeatureAsMemberInClassesDecl and FeatureAsMemberInInterfacesDecl respectively. In JastAdd, a block is treated as a statement of statements, as shown in Figure 4.6 and expressed in Lines 5 and 6 in Listing 4.14. To represent a statement or set of statements as well as a block to be part a feature, we use the FeatureAsMemberInMethodDecl class.

Listing 4.14: Representing Feature Entities based on Containment Hierarchy

```
1 FeatureAsMemberInClassesDecl : BodyDecl ::=
2    featureAsMemberOfClass:FeatureContainment;
3 FeatureAsMemberInInterfacesDecl : BodyDecl ::=
4    featureAsMemberOfInterface:FeatureContainment;
5 abstract Stmt;
6 Block : Stmt ::= Stmt*;
7 FeatureAsMemberInMethodDecl : Stmt ::=
8    featureAsMemberOfMethod:FeatureContainment;
```

With these three classes it is possible to represent one or more body declarations inside a class (including fields, methods, constructors, and inner classes/interfaces, etc.), one or more body declarations inside an interfaces (including constants, method signatures, inner classes/interfaces, etc.), and one or more statements, and blocks inside methods, constructors, and block statements (such as for and while) to be part of features thus covering a range of granularity from entire classes to individual statements. In order to indicate that a class itself is part of a feature, we use the convention of containing all its body declarations inside that feature. We show the concrete syntax for all the above constructs in the next section.

4.2.2 Syntactic Extension

In order to simplify the discussion of the domain-specific concrete syntax adopted by FeatureJ, we make use of an example SPL. This is an SPL of notepad applications. It is based on the idea that variants of notepad application can be generated by selecting various formatting features. It is illustrated in Figure 4.7. It consists of all relations and operators generally found in a feature model [Kang et al. 1990].

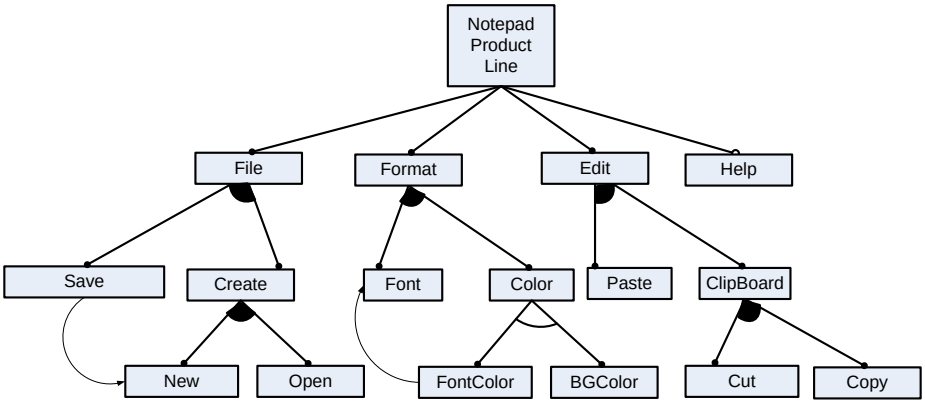

Figure 4.7: Notepad Product Line

As discussed in the previous section, an SPL is represented by the node FeaturePLDecl. At the level of syntax, the Notepad Product Line (NPL) is represented as shown in Listing 4.15. A productline type is used to represent the FeaturePLDecl at the level of syntax (similar to using `class` for a ClassDecl). The productline type identifier signifies the actual SPL being mirrored at the syntactic level.

A productline type consists of features and constraints blocks in which features and constraints of the base feature model are enlisted. The relations between a single parent node and its children is specified in each expression inside the features block. The all, more, and one keywords signify the and, or, and alternative relations between the children of a given node. The optionality of a feature is indicated by a question mark '?'. Altogether, the productline type notepadPL contains 16 features which mirror the features and the relations and constraints between them shown in Figure 4.7. The top level features and the relation between them are enlisted outside the features block. This is followed by the constraints block in which, inclusion and exclusion constraints in the base feature model can be specified using special operators <–> and >–< respectively.

The selection of features based on the relations and constraints is represented by the node FeaturePLVariantDecl as discussed in the preceding section. At

Listing 4.15: FeatureJ Syntax for productline Type Definition

```
productline notepadPL {
  features {
   File : more(Create, Save),
   Create: more(New, Open);
   Edit : more (ClipBoard, Paste),
   ClipBoard: more(Cut, Copy),
   Format : more(Font, Color),
   Color : one (FontColor, BGColor),
   Help
  }
  all(File, Edit, Format, Help?)
  constraints {
   FontColor  <-> Font,
   Save <-> New
  }
};
```

the level of syntax, it is indicated by a variant type. A variant type is always defined within the context of a productline type by specifying productline type name along with the variant type name.

Listing 4.16: FeatureJ Syntax for variant Type Definition - I

```
variant notepadPL simpleNotepad {
  File  = [Create and Save],
  Create = [New],
  Edit = [ClipBoard],
  ClipBoard = [Cut and Copy],
  Format = [Font],
  Help
};
```

With this convention, it is indicated that the variant type simpleNotepad is a variant of the SPL represented by the productline type notepadPL. A variant type essentially indicates selection between the children for each parent node in the base feature model. Instead of using a comma to indicate the selected children, we use the keyword and for the same. This mirrors the natural way of selecting features, such as when a user wants to say that he needs to select features b *and* c from a parent feature a. The equality operator '=' is used to indicate which children a parent feature consists of in this variant. The simpleNotepad variant type thus consist of 11 out of 16 features from the notepadPL productline type as shown in Listing 4.16.

Similarly, Listing 4.17 shows another variant type colorNotepad of the product-

Listing 4.17: FeatureJ Syntax for variant Type Definition - II

```
variant notepadPL colorNotepad {
 File  = [Create],
 Create = [New],
 Edit = [ClipBoard],
 ClipBoard = [Cut and Copy],
 Format = [Color],
 Color= [FontColor]
};
```

line type notepadPL consisting of 10 out of 16 features in the base feature model. Any other variant type definition indicating another selection from the base feature model represented by the productline type notepadPL can be expressed in the same manner.

While the productline and variant types indicate an SPL and its variants, the code fragments in this application, that belong to one of the features specified in the features block are expressed using the feature type definition as shown in Listing 4.18.

Listing 4.18: A Feature Containing an Inner Class inside a Class

```
public class Notepad extends JFrame {
 feature notepadPL Cut {
  class EditCut implements ActionListener {
   public void actionPerformed(ActionEvent e) {
    String selText=text.getSelectedText();
    if(selText==null)
    return;
    StringSelection stringSel=
     new StringSelection(selText);
    board.setContents(stringSel,stringSel);
    text.replaceSelection("");
   }
  }
 }
 ...
}
```

We use the convention that while the features of a productline type are declared in the features block, their definitions are represented by a feature type definition.

A feature type definition is represented internally by the nodes FeatureAsMemberInClassesDecl, FeatureAsMemberInInterfacesDecl, and FeatureAsMemberInMethodDecl as discussed in the preceding section. All three nodes represent

a feature type at a varying granularity as discussed earlier. Accordingly, Listing 4.18 shows the definition of feature type Cut that contains a body declaration (in this case an inner class that implements the ActionListener interface to emulate cut operation) inside the Notepad class.

Listing 4.19 shows another definition of the feature type Cut. Notice that while a feature of a productline is declared once, definitions for it can be provided as many times as required, to assign all code fragments that contain the functionality expressed by the feature type. Furthermore, a feature type is always defined within the context of productline type, which must be the same as the productline type where this feature was declared. Other definitions for the Cut feature as well as other feature types are similarly defined in the application classes of the notepad application.

Listing 4.19: A Feature Containing Statements in a Constructor

```
1 public class Notepad extends JFrame {
2  private JMenuItem cut;
3  public Notepad() {
4   feature notepadPL Cut {
5    cut=new JMenuItem("Cut Ctrl+X");
6   }
7  ...
8  }
9 }
```

Both the productline and variant type definitions occur as body declarations, which means that they must be defined inside a class. It must be noted that like simpleNotepad and colorNotepad, many other variant types may be defined in a single program. Each variant type essentially denotes a version of the complete notepad application with application specific classes based on the selected features. In order to invoke a method of any of the application classes, i.e., Notepad, TextFilter, and NotepadGUI classes which make up the notepad application, the syntax shown in Listing 4.20 can be used.

Line 12 in Listing 4.20 shows that to call the method run() of the NotepadGUI class of the variant type simpleNotepad, it is indicated that the variant type is a variant of the productline type notepadPL and furthermore, the NotepadGUI is a class with its specific version from the variant type simpleNotepad. With this syntax an instance of NotepadGUI is obtained and method run() is called on it. Alternatively, an identifier for the instance can be specified instead of directly specifying method name. In this case, the instance can be used to call specific methods with regular Java syntax.

Considering that the NotepadGUI class of the variant simpleNotepad has a run() method as shown in Listing 4.21, upon its execution a notepad GUI application starts that is the simpleNotepad variant.

Listing 4.20: FeatureJ Syntax for Generating and Executing a variant

```
1  public class Launcher{
2   productline  notepadPL {
3   ...
4   };
5   variant notepadPL simpleNotepad {
6   ...
7   };
8   variant notepadPL colorNotepad {
9   ...
10  };
11  public static void main(String args[]) {
12   notepadPL::simpleNotepad NotepadGUI->run();
13   notepadPL::colorNotepad NotepadGUI->run();
14  }
15 }
```

Listing 4.21: Method to Initialize the Notepad

```
1  public class NotepadGUI {
2   public void run(){
3    Notepad n=new Notepad();
4    n.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5    n.setSize(300,200);
6    n.setVisible(true);
7    }
8  }
```

Having acquainted ourselves with the concrete syntax, in the next section, we elaborate how we support the domain-specific checking in FeatureJ.

4.2.3 Name/Type Analysis and Error Checking in FeatureJ

As discussed earlier on page 61, the name analysis in Java aims at establishing the visibility and scope of various Java entities and ascertaining the meaning of names as they appear in different syntactic contexts, and on page 62, the type analysis in Java consists of computing the type of an expression and determining subtype relations. From the perspective of productline, variant, and feature types, we need to establish their visibility and determine the interrelationship between them.

We use a combination of AST node attributes and an external data structure called FeatureDeclsCollection to store all information related to productline type as well as the variant and feature types that are defined in its context. Note

that the purpose of the external data structure is to store information about feature domain entities in an interconnected manner. Every feature type knows which productline it belongs to and its relative position within the feature model of this productline type. Similarly, every variant type knows which productline type it belongs to and productline type keeps record of how many variant types are defined based on it. We make a design choice that as soon as the nodes representing these types are visited, the external data structure is populated with all the relevant information using specialized attributes and these nodes are rewritten so that latter traversals of the AST see pure Java AST while all the information related to feature domain entities is preserved and is available to use separately. Toward this end, we use JastAdd's rewrite mechanism. We elaborate this in the following.

Name/Type Analysis of Feature Domain Entities

As the variant and feature types are defined within the context of a productline type, we need to establish the visibility and scope of this productline type and with it, we can ascertain the meaning of the names of the variant and feature types defined within its context.

Listing 4.22: Abstract Grammar for productline and variant Types inside Classes

```
1 abstract FeatureMember : BodyDecl;
2 LocalFPLDeclMember : FeatureMember ::=
3  productLineDecl:FeaturePLorVariantDecl;
4 LocalFVariantDeclMember : FeatureMember ::=
5  variantDecl:FeaturePLorVariantDecl;
```

The actual definitions of productline and variant types in FeatureJ take place inside a class, as body declarations. These body declarations are represented using the AST node classes LocalFPLDeclMember and LocalFVariantDeclMember as shown in Listing 4.22.

The class LocalFPLDeclMember indicates a productline type definition that occurs inside a class. This node is rewritten as shown in Listing 4.23, so that the complete structure of the productline type is stored in the external data structure. Using this information the productline type is scoped in such a way that it is visible application-wide. We do not restrict the scope of the productline type to a class or a package, because it is referred in both the variant and feature type definitions which can occur across classes and packages of the same application.

The class LocalFVariantDeclMember indicates a variant type definition that occurs inside a class, which is rewritten so that the complete structure of the variant is stored in the external data structure. A variant type is scoped inside

Listing 4.23: Using Rewrites to Transform a productline Type

```
rewrite LocalFPLDeclMember  {
 to EmptyBodyRewrite {
  //init FeatureDeclsCollection fdcl
  ...
  //populate product line information
  FeaturePLDecl fpdl=(FeaturePLDecl)getproductLineDecl();
  fpdl.setFeaturebag();
  fdcl.getFeatureDeclsMap().
   put(fpdl.name(), fpdl.getFeaturebag());
  fdcl.getPlConstraintMap().
   put(fpdl.name(), fpdl.constraints());
  return new EmptyBodyRewrite();
  }
}
```

the class where it is declared and can be referred to in other classes using an import statement that specifies the class of this variant type.

The rewrites are also applied to FeatureAsMemberInClassesDecl, FeatureAsMemberInInterfacesDecl, and FeatureAsMemberInMethodDecl nodes that represent feature type definitions inside a class, an interface, and methods, constructors, block structures respectively. Listing 4.24 shows how a FeatureAsMemberInClassesDecl node is rewritten that contains exactly one body declaration.

Listing 4.24: Using Rewrites to Transform a feature Type Definition in a Class

```
rewrite FeatureAsMemberInClassesDecl {
 when(((featureAsMemberInClassesCont)
  getfeatureAsMemberOfClass()).getNummemberBodies() == 1)
 to BodyDecl {
   //init features that contain this body declaration
   ...
   //set feature containment recursively
   featureAsMemberInClassesCont.
    getmemberBodies(0).setWrapperFeatures(features);
   featureAsMemberInClassesCont.
     getmemberBodies(0).visitFineGNodesToWrap(features);
   return featureAsMemberInClassesCont.getmemberBodies(0);
   }
}
```

When rewriting a feature type definition in a class, for each AST node that is contained within the feature type definition, the name of the feature is stored in the AST node. This is done recursively. For instance, recall that in Listing

4.18, we showed an inner class EditCut contained in the definition of feature type Cut. When rewriting this feature type definition, the AST node representing the inner class EditCut as well as all the AST nodes representing its contents are attributed with the name of feature type which is Cut. The same process is applied when rewriting FeatureAsMemberInInterfacesDecl and FeatureAsMemberInMethodDecl nodes. In each case the information about the feature types is distributed recursively to contents of body declarations and blocks contained in them.

Syntax for Multiple, Nested, and Alternative Feature Containments

Since the FeatureJ implementation of the NPL does not contain situations such as multiple features containing same code fragment, nested feature containments, and alternative feature containments we describe the syntax and the treatment for the same using a general example as shown in Figure 4.8.

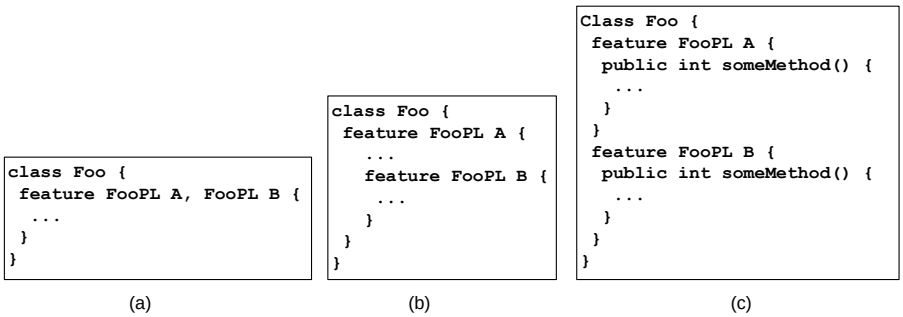

Figure 4.8: Containments for Multiple, Nested, and Alternative feature types

When multiple feature types contain same code fragments as shown in Figure 4.8 (a) in which feature A and feature B of productline FooPL, the code fragments are included in avariant of FooPL when at least one feature is selected in that variant among these features.

When feature containments are nested as shown in Figure 4.8 (b), code fragments of feature B are included in a variant only if feature A is also selected along with feature B in that variant. Any level of nesting is possible because rewrites of FeatureAsMemberInClassesDecl, FeatureAsMemberInInterfacesDecl, and FeatureAsMemberInMethodDecl are carried out at the moment they are accessed thus rewriting of these AST nodes operates from outermost level to the innermost.

Two unique situations may occur that may lead to compile-time errors:

1. As shown in Figure 4.8 (c) feature A and feature B contain the same method but possibly different implementations. If feature A and feature

B are not indicated to be alternative in the definition of productline FooPL and they get selected in a variant then method someMethod() is multiply defined. If feature A and feature B are indicated to be alternative in the definition of productline FooPL but still selected in a variant then this variant is not valid according to productline FooPL.

2. In Figure 4.8 (b), feature B is selected but not feature A in a variant. This will result in not including the code fragments in feature B because A is not selected. If this is not the desired behavior, it may lead to incomplete implementation and may require reassessing the containments or the design of the code base of productline FooPL.

In both these cases, FeatureJ provides support in the form of checks that a variant is valid and no compile-time error may occur in the program to be generated that represents this variant. After applying rewrites to all productline, variant, and feature types in a FeatureJ application, we get the program AST that represents the original Java application, one or more SPLs of which are to be constructed, and the external data structure. The information in the external data structure is used for implementing domain-specific error checking while the information about which feature a given AST node belongs to is stored in the AST is used in generating variants. We explain the domain-specific error checking in the next section, followed by the architecture used to generate variants.

Error Checking in FeatureJ

The error checking in FeatureJ consists of two steps. First, for each variant type, determine it is a valid variant based on the productline type. This ensures that the selection of feature types from the productline type is according to the relations and constraints between the feature types that leads to the definition of this variant type. Second, the information stored in the AST nodes about which feature types contain them, is used in relating possible compilation errors to these features. We explain both in the following.

The information stored in the external data structure mimics the tree structure of the productline and variant types with parent-child relationships between various feature types. With this information, it is possible to check that a variant type is a valid variant according to various relations and constraints between the feature types declared inside the productline type. For instance, consider the variant type simpleNotepad in Listing 4.16 which is based on the productline type notepadPL shown in Listing 4.15. The feature types Create and Save are selected from the parent feature type File. This is checked against the more operator specified in productline type notepadPL for the feature type File. The more operator is implemented as a method that checks that one or more of the feature type File's children are selected. Furthermore, the feature type Save requires the feature type New, according to the first constraint in the

constraints block of the notepadPL productline type. Like relation operators, constraints are checked through a method that ensures that if the feature type T that requires feature type T' is selected in a variant type, then the feature type T' must also be selected otherwise a compilation error is reported.

In order to ensure that a valid variant of an SPL in FeatureJ would compile without errors, we extend the error checking methods in JastAdd for all AST nodes. Since a FeatureJ application can contain multiple productline and variant type definitions, we choose to consider this situation when implementing the error checking in FeatureJ. We separate the errors in FeatureJ to variant-specific and productline-specific errors. We basically distinguish between those code fragments in a FeatureJ program that are contained in features and those that are not.

The rationale behind this distinction is that when a compilation error originates in the code fragments that are contained in one or more features, these errors will be present in the actual variants to be generated based on variant types in which these features are selected. Such compilation errors are referred to as variant-specific errors. These errors are reported by specifying the names of those variant types along with the features selected in them that contain code fragments where the compilation errors originated.

Furthermore, note that not all code fragments in an application are contained in features. The code fragments that are not contained in any features can be assumed to make up the base program, i.e., these code fragments will be present in any variant to be generated irrespective of which features are selected in the corresponding variant types. The compilation errors that originate in the base program are referred to as productline-specific errors. These errors are reported by specifying the name of the productline type and location.

When an error originates in code fragments contained in one or more features, but none of these features are selected in any of the variant types defined in the FeatureJ program, then such errors are ignored. Again, the rationale behind our design in error reporting is that such errors will not affect the proper execution of any variant types defined in the program and therefore they need not be considered. In the current implementation of FeatureJ, errors of this kind are not reported.

Recall that for each AST node class in JastAdd, implementation of one or more of the error checking methods are provided depending on corresponding JLS specifications [Gosling et al. 2005] (see the aspect ErrorCheck 4.7 on page 64 and examples of error checking methods 4.8 to 4.10 on pages 65–66 for AST node class ClassInstanceExpr). We need to extend these error checking methods in such a way that we can distinguish between and report the productline- and variant-specific errors separately as they occur. For this, we use the information in each AST node about the features it is contained in as discussed on page 82 (In the explanation of Listing 4.24). To get the information about productline type these features are declared in and which variant types they are selected in, we use the information stored in the external data structure as

Listing 4.25: Reporting variant-specific NameChecking Errors for ClassInstanceExpr

```
1  aspect FeatureSCErrorReporting {
2   public void ClassInstanceExpr.nameCheck_ContainedSelected
3    (String variantID,ArrayList<String> inFeatures) {
4    super.nameCheck();
5    if(decls().isEmpty())
6     error("In the variant " +  variantName + " " +
7      "can not instantiate " + type().typeName() +
8       " no matching constructor found in "  +
9         type().typeName() + " in " + inFeatures);
10   else if(decls().size() > 1 && validArgs()) {
11    error("In the variant " +  variantName + " " +
12      "several most specific constructors found");
13    for(Iterator iter = decls().iterator();
14      iter.hasNext(); ) {
15       error( "   " + ((ConstructorDecl)iter.next()).↩
             signature()
16         + " in " + inFeatures);
17    }
18   }
19  }
20 }
```

discussed on page 80. We subject the program AST of a FeatureJ program to error checking through a method called featureSafelyComposeCheck() whose responsibility is to separate the calls to error reporting methods based on our distinction between feature contained and non-contained code fragments and consequently errors that are variant-specific and productline-specific.

The error() method in JastAdd adds the error message to a list stored in the Program node and adds error messages originating in any AST nodes to this list. In order to customize the error reporting, we create alternate methods for each AST node that specify whether the error message should indicate it is a variant-specific or productline-specific error. Listing 4.25 shows the nameCheck_ContainedSelected() method of ClassInstanceExpr. It is called for a given ClassInstanceExpr node when the node is contained in one or more features that are selected in the variant type whose name is `variantID`. If the node contains an error, then variant-specific errors are recorded as indicated in Lines 6-9 and 11-16. This also enables us to specify which features are broken as shown in Lines 9 and 16. The default implementation of the error() method in JastAdd also adds the location information. Additional versions of nameCheck() method include one method that reports errors in nodes that are contained but are part of features that are not selected in a variant and another method in which a node is not contained in

a features thus making it a productline-specific errors. All AST nodes contain these three versions of methods that make up error checking in JastAdd. For instance, in the node ClassInstanceExpr, *_ContainedSelected(), *_ContainedNotSelected(), and *_NotContained() methods are included for FeatureJ error reporting where * is each one of the nameCheck(), typeCheck(), and accessControl() methods.

In this way, using the information stored in the external data structure, FeatureJ is able to specify informative error messages that include the kind of error, which features contain the node where the error originates and the location in the original FeatureJ source files.

4.3 Architecture for First-class Features in FeatureJ

In the preceding sections, we explained the FeatureJ language internals including the syntactic and semantic extensions of the Java compiler in JastAdd that enable integrating domain-specific concrete syntax and domain-specific checking related to feature domain entities. A consequence of a first-class representation of feature domain entities is that multiple productline and variant types may be defined and referred to at the same time. In case of Java, this has important implications. Given a FeatureJ application, which is basically a Java application with definitions of one or more productline, variant and feature types, generating multiple variants implies that there will be different versions of the same set of classes of the application for each variant to be generated. This situation is depicted in Figure 4.9.

In order to address multiple versions of the same set of Java classes, we need to overcome the hurdle of Java class-loading conflicts, which we elaborate next.

4.3.1 JVM and Class-loading

In Java, an application is executed using Java Virtual Machine (JVM). A JVM is an abstract computing machine. Similar to real machines, it has an instruction set and enables manipulation of various memory areas at run-time. A JVM provides platform-independent method of executing Java programs. It knows nothing of the Java programs it executes, rather it understands only the .class file format. The .class file format is hardware- and operating system-indepependent binary format. A .class file contains JVM instructions also called byte-codes and a symbol table with other auxiliary information necessary for execution of a Java program [Lindholm and Yellin 1999]. The Java class-loading in JVM is a process of finding the binary representation of a class or interface type with a specific name and creating a class or interface from that binary representation. Such a class or interface is introduced into the run-time state of the JVM, i.e., linked, so that it can be executed. The process of Java

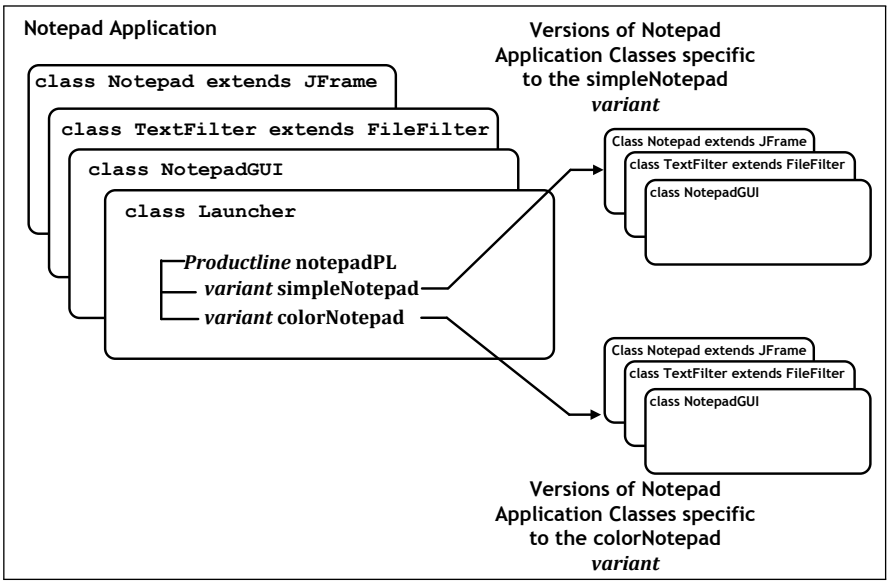

Figure 4.9: Multiple Variants in a FeatureJ Program

class-loading is highly idiosyncratic. Particularly, there are following notable aspects of Java class-loading:

Class loaders In JVM, each and every class is loaded by an instance of java.lang.ClassLoader class and its subclasses. This class uses a delegation model to search classes and resources. Each instance of a ClassLoader class has a parent class loader. When locating classes and resources, a ClassLoader instance delegates the search to its parent class loader before attempting to find the class or resource itself. The class-loading process begins with the bootstrap class loader which loads the classes of Java platform including classes in rt.jar. This is followed by extension classes which are located as .jar files in the Java extensions directory. Finally, the classes defined by developers and located in other .jar files are loaded using the classpath environment variable. When a class is loaded, all classes it references are loaded recursively. A class is loaded once and then cached by the JVM to ensure that its byte-code does not change.

Class/Interface Names and Namespaces In the JVM, a class type is uniquely determined by the combination of the class name and the class loader. The execution of Java byte-codes can be visualized as a set of classes partitioned into separate namespaces [Gosling and McGilton 1996]. The Java security features ensure that when a class is loaded from a location either from a local file system or a network location, it is placed into its private namespace associated with its origin. When a class ref-

erences another class, it is first looked up in the namespace of the local system and then the namespace of the referencing class. Classes imported from different places are separated from each other. Using the delegating class loaders makes it possible to maintain namespace separation while sharing a common set of classes [Liang and Bracha 1998].

From the situation shown in Figure 4.9, it is clear that we need to take care of multiple versions of the same set of SPL classes. Based on the discussion above, we employ a solution in which we generate variant-specific classes in different locations on the file system and load them when required using separate class loaders using Java reflection. Each variant type is associated with an AST which represents feature selection specific to it. All variant-specific ASTs are obtained from the main program AST and transformed to individual sets of .class files using the JastAdd backend.

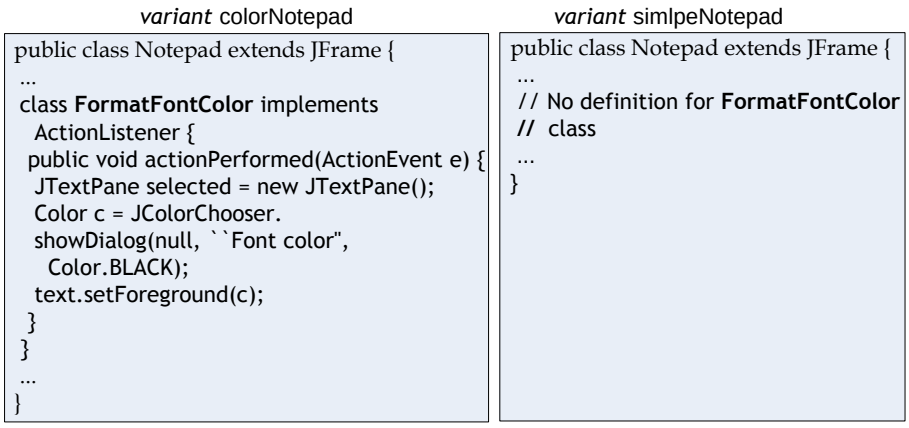

Figure 4.10: Different Notepad Classes per variant Type

For instance, consider the versions of class `Notepad` from the simpleNotepad variant type and colorNotepad variant type. While the Notepad class that is part of variant simpleNotepad does not consist of the inner class FormatFont-Color, the Notepad class that is part of variant type colorNotepad does. The variant-specific classes are generated by first creating an AST that represents the variant type. This is illustrated in Figure 4.10.

These two classes and all other classes that comprise the variant types defined in a FeatureJ program are generated in different folders on the file system where a FeatureJ program is being executed. The default directory used by FeatureJ for this purpose is the *temp* directory of the operating system. Inside the *temp* directory, FeatureJ creates folders that correspond to variant type names. For instance, in the ongoing example, two directories called simpleNotepad and colorNotepad are created under the *temp* directory. Under each of these directories, a subdirectory structure is created that corresponds to the package

hierarchy of the original application, in this case the Notepad application. Various classes pertaining to a variant are placed according to their locations in the package.

4.3.2 Variant Composition and Generation Architecture in FeatureJ

Notice that we have excluded the Launcher class from the variant-specific application classes in Figure 4.9. The Launcher class is what we call an entry point for a FeatureJ application. After starting a FeatureJ application with the Launcher class specified as an application main, internally, FeatureJ launches another process to actually execute the application with the variant types defined in it.

The responsibility of loading variant-specific classes as well as obtaining instances of these classes is given to a class called `PLVariant`. Similarly, two other classes `PL` and `Feature` provide access to the underlying productline and feature types. Since data can no be shared between processes, we re-parse the main program and obtain the main AST once at run-time irrespective of the number of productline and variant types defined. The structural information stored in the external data structure is made available to specific instances of the `PL`, `PLVariant`, and `Feature` classes based on which productline, variant, and feature types they are used to represent. We call classes `PL`, `PLVariant`, and `Feature` as meta-classes in the sense that they are used to manipulate underlying productline, variant, and feature types respectively, as we explain shortly.

FeatureJ application initiation by executing the application main and deploying various variants using a separate process are the responsibilities of the FeatureJ application container. The application container acts as platform from which variants are generated and executed. This is illustrated in Figure 4.11. The arrows in Figure 4.11 indicate general flow of control. With this design, it is possible to differentiate between variants generated by composing features at compile-time and at run-time as described below:

Variant composition at compile-time When a variant is composed statically, the variant AST is generated after creating a folder hierarchy for its .class files as discussed earlier. The .class files generated from the AST are stored in the PLVariant instance for this variant type as a pair of qualified class/interface name and a class Class object obtained by loading the .class file. An object of any SPL class is obtained using reflection and can be used as a regular object even though it is variant-specific.
An excerpt of the code shown in Listing 4.20 is presented in Listing 4.26. Using rewrites as discussed earlier in Section 4.2, this syntax is transformed into an equivalent code shown in Listing 4.27. Note that Line 3 in Listing 4.26 obtains a NotepadGUI object based on a constructor with no arguments. If the constructor of the application class whose objects

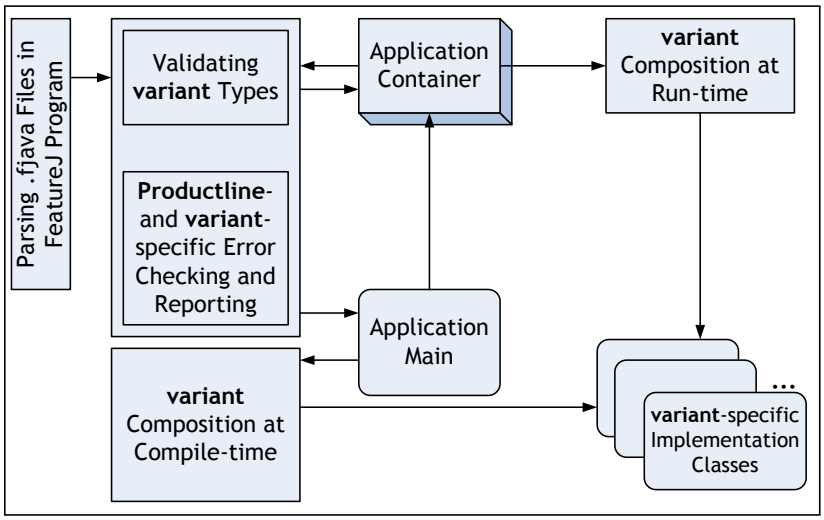

Figure 4.11: FeatureJ Architecture for Variant Composition

Listing 4.26: Original variant and Application Class Instance Method Call

```
1 public class Launcher {
2  public static void main(String args[]) {
3   notepadPL::simpleNotepad NotepadGUI->run();
4   }
5 }
```

are to be obtained has parameters, they can be specified in parentheses following application class name in Line 3 and they are included in the rewriting process and later when getting objects via reflection in Line 6 of Listing 4.27.

Listing 4.27: Transformed Syntax for variant simpleNotepad

```
1 public class Launcher {
2  public static void main(String args[]) {
3   PL NotepadPL= new PL("notepadPL");
4   PLVariant simpleNotepad=
5    new PLVariant(NotepadPL,"simpleNotepad");
6   simpleNotepad.call("NotepadGUI","run");
7   ...
8  }
9 }
```

Variant composition at run-time A variant in FeatureJ that is generated

at compile-time can be modified at run-time by adding or removing feature types from it. This is results in a new variant. The addition and removal of features from a variant is carried out using PLVariant class methods add() and remove() respectively. Recall that a PLVariant class is associated with a PL instance that represents the productline type at run-time. The PL instance contains the original AST which is used to affect modifications to the current PLVariant instance. Internally, a new AST is generated that reflects the changes. Following this, similar process is followed for generating .class files from the AST and loading and mapping Class objects as in compile-time generation. Instead of compiltime, a new PLVariant instance is obtained at run-time that represents the modified variant type. This variant modification syntax shown in Listing 4.28 is transformed using JastAdd rewrites to syntax that makes use of the PLVariant class's add() method with which one or more feature types can be added to a variant. The modified variant is referred with a different name as simpleNotepadModified. This variant contains all feature types selected in the simpleNotepad variant as well as feature types Color and FontColor. When returning a modified variant type, it is checked for being valid and without errors as discussed in Section 4.2.3 on page 84.

Listing 4.28: Adding Features to a variant type

```
1 public class Launcher {
2  public static void main(String args[]) {
3   notepadPL::simpleNotepad
4    ->add(FontColor)>>simpleNotepadModified;
5   notepadPL:: simpleNotepadModified NotepadGUI->run();
6   }
7 }
```

This kind of design enables us to provide access to various feature domain entities at different times in program execution, without having to represent information related to feature domain entities in the byte-code and later use byte-code instrumentation tools to gain access to these entities.

The public methods of class `PL` include methods like allFeatures(), mandatoryFeatures(), optionalFeatures() getVariantList() whose purpose is clear in their names. It also contains a method called getFeature(String name) that returns a `Feature` object of named feature. The public methods of `PLVariant` include getVariantClass(String className) that returns the Class of the named application class, getVariantObject(String className) that returns an object of the named application class and selectedFeatures() which returns names of features selected in the underlying variant types. The `Feature` class similarly contains methods for getting the name of the feature, and whether its

mandatory or optional. Both `PLVariant` and `Feature` classes contain a getParentPL() method that returns the parent PL object. These classes contain many protected methods used internally for purposes such as calling specific methods of application class objects as shown in Listing 4.27 and when adding features to a variant as shown in Listing 4.28. Note that a FeatureJ program may contain statements for composition of a variant type with the syntax as shown in Listings 4.26 and 4.28 as well as statements using `PL`, `PLVariant`, and `Feature` classes as shown in Listing 4.27 at the same time. Furthermore, objects of `PL`, `PLVariant`, and `Feature` classes can be passed as parameters to a method and can be returned from a method just like regular Java objects.

Differentiating Adaptation from Composition at Run-time

One important point to note is that the state of objects from the original variant type is not preserved when modifying a variant. That is, while a modified variant is generated by run-time composition of features in the underlying AST, the run-time adaptation is not supported. In order to use objects of SPL classes of this variant, they have to be obtained anew. Therefore, when variant simpleNotepad is modified to obtain simpleNotepadModified, an instance of NotepadGUI is obtained again and run() is called. If run-time adaptation were supported then the NotepadGUI class of simpleNotepad variant would be updated to support additional feature FontColor.

4.3.3 FeatureJ Compiler for Java 1.4

We described on page 70, how the capabilities provided by JastAdd are used when implementing a Java language extension. We asserted that extending Java types, extending the name/type analysis, and using rewrites to transform extension source to Java are the three often-used ways of extending Java using JastAdd. We use each of these capabilities when implementing FeatureJ atop JastAdd Java 1.4 compiler. The syntactic and semantic components are structured as illustrated in Figure 4.12. First, we provide the domain-specific abstractions in FeatureJ by extending JastAdd representation of Java types. Second, the domain-specific concrete syntax and checking in FeatureJ is supported by extending JastAdd implementation of Java name/type analysis and error checking. And third, we use JastAdd's rewrite system for transforming the domain-specific concrete syntax when initially storing information related to various feature domain entities and later in expressing variant generation and modification in terms of meta-classes `PL`, `PLVariant`, and `Feature`.

Java 1.4 lexical and syntactic components are extended using FeatureJ-specific lexer and parser written in JFlex and Beaver parser generator respectively. FeatureJ-specific semantic analyses are arranged in various .jrag files. Meta-classes and external data structures for storing information about feature do-

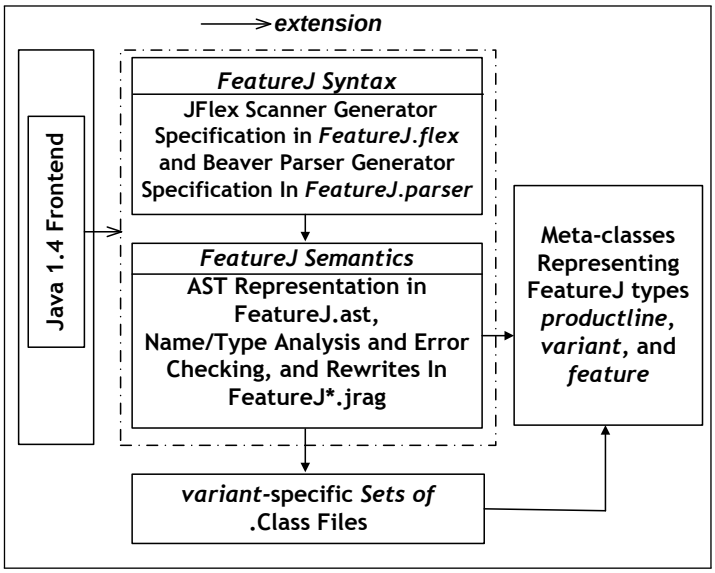

Figure 4.12: Complete Structure of FeatureJ Compiler using JastAdd

main entities are implemented separately as a library of classes. The arrows with filled triangle indicate how each component is used.

Our decision to implement the meta-classes and the external data structures separate from AST is taken on the basis that it enables us to accommodate any extensions to base Java 1.4 compiler. Whenever adding extensions to Java 1.4 compiler, the architecture explained earlier in Section 4.3.2 on page 90 remains untouched. We utilize this facet of the FeatureJ architecture in extending it to accommodate JastAdd's Java 1.5 extension of Java 1.4 compiler.

4.3.4 Extending FeatureJ to Support Java 1.5

Earlier in Listings 4.11 to 4.12 on pages 68–69, we showed that JastAdd implements Java 1.5 compiler atop Java 1.4 compiler by (a) adding as required, new AST nodes that represent Java 1.5 extensions using abstract grammar, (b) refining pre-existing analyses when needed, and (c) using rewrites to generate classes when required, e.g. for enum types as well as to replace concrete syntax with intended semantics, e.g., internally representing enhanced for statements in Java 1.5 in terms of Java 1.4 for statement syntax.

All the new AST nodes added in Java 1.5 compiler extension of Java 1.4 compiler are subclasses of existing nodes. This means that FeatureJ does not have to take care of Java 1.5 syntax additions for representing feature containments in Java 1.5, because parsing of nodes representing extensions returns superclasses by default. For instance, generic type declarations in Java 1.5 are

represented by GenericClassDecl class which is a subclass of ClassDecl. Listing 4.29 shows its declaration in the abstract grammar.

Listing 4.29: Abstract Grammar for Generic Classes in Java 1.5

```
1 GenericClassDecl : ClassDecl ::= Modifiers <ID:String>
2  [SuperClassAccess:Access] Implements:Access* BodyDecl*
3  TypeParameter:TypeVariable* /ParTypeDecl:ParClassDecl*/;
```

Note that ParClassDecl indicates a parameterized type. Rest of the nodes are part of the Java 1.4 specification upon which abstract grammar of Java 1.5 is built. When parsing a generic type declaration, a ClassDecl node is returned as shown in Listing 4.30.

Listing 4.30: Parser Grammar for Generic Classes in Java 1.5

```
1 ClassDecl class_declaration =
2  modifiers.m? CLASS IDENTIFIER type_parameters.p super.s?
3  interfaces.i? class_body.b
4  {:
5    return new GenericClassDecl(new Modifiers(m),
6    IDENTIFIER, s, i, b, p);
7  :};
```

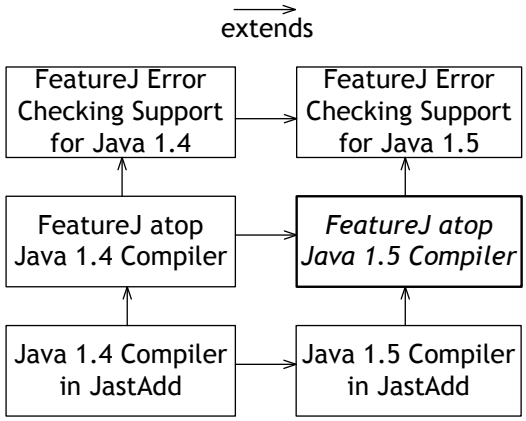

Figure 4.13: Supporting Java 1.5 with FeatureJ

Consequently, FeatureJ does not have to account for the syntax of generic class declarations when representing feature containment, such as e.g., feature containment of a generic inner class declaration. The amendments to FeatureJ compiler for Java 1.4 are therefore restricted to (a) subsuming the semantic

analyses of newly added AST nodes in FeatureJ name/type analysis and error checking and (b) accounting for refined pre-existing analyses.

The semantic analyses of newly added nodes are subsumed by providing an implementation of the featureSafelyComposeCheck() method for them which is carried out in a manner similar to explained earlier (cf. Error Checking in FeatureJ in Section 4.2.3 on page 84). To account for the refined pre-existing analyses, the featureSafelyComposeCheck() method refers to the refined version of an attribute instead of the original one. For instance, name checking of ClassInstanceExpr which is refined to accommodate enum types as shown in Listing 4.11 on page 68, is considered inside the featureSafelyComposeCheck() method for ClassInstanceExpr in the the Java 1.5 extension of the FeatureJ compiler when coding various versions that report variant- and productline-specific errors. Treatment of (a) and (b) thus give rise to structure shown in Figure 4.13 for extension to FeatureJ supporting Java 1.5 source. Note that *FeatureJ atop Java 1.5 compiler* component is obtained automatically as explained in the discussion above and only error checking for Java 1.5 source is extended.

Note that the FeatureJ architecture shown in Figure 4.11 on page 91 is not affected by the extension to FeatureJ to support Java 1.5 source because information related to feature domain entities and meta-classes is stored separately from AST which is extended in Java 1.5 and the changes are accommodated by FeatureJ extension to support Java 1.5 source.

4.3.5 Implementation Statistics

The SLOC (source lines of code) of FeatureJ implementation atop the JastAdd Java 1.4 and 1.5 compilers is shown in Table 4.1. It shows the SLOC in .jrag specification of Java 1.4 and 1.5 compilers and FeatureJ implementation. Figure 4.14 shows the folder hierarchy of Java 1.4 and 1.5 compilers. JastAdd generates Java code using the AST (.jrag files), parser (.parser files), and scanner (.flex files) specifications in the AST, parser, and scanner folders which consist of Java classes of all AST nodes and parser and scanner for the designated compiler respectively.

Compiler Module	JastAdd SLOC	FeatureJ SLOC	Compiler
Base Compiler	6K	2K	1.4
Error Checking	3K	4K	1.4
Base Compiler + Error Checking	6K	2K	1.5

Table 4.1: FeatureJ Implementation

All FeatureJ-specific components that extend the AST are arranged in the FeatureJ folder including the FeatureJ.ast file for AST specification, FeatureJ.parser

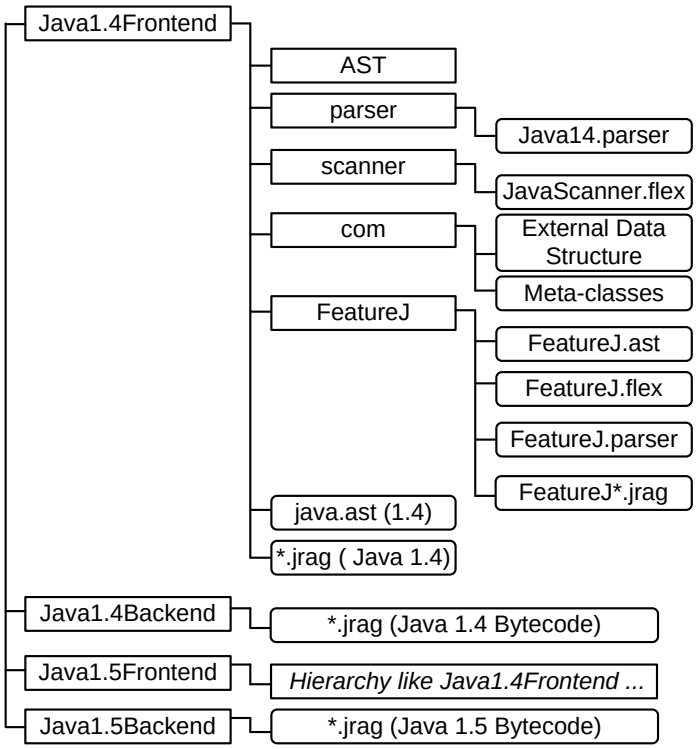

Figure 4.14: Folder Hierarchy of FeatureJ Implementation using JastAdd

for parser specification, and FeatureJ.flex for lexer/scanner specification. The meta-classes and the FeatureJ compiler frontend are not part of the AST rather they use and manipulate the AST node classes in the AST folder. They are therefore separated out into another folder called com.

An ant build file is used to direct the composition of specific kinds of language components. Thus Java 1.4 and FeatureJ parser specifications are merged. Similarly Java 1.4 and FeatureJ lexer specification are combined. Then using Java 1.4 and FeatureJ AST specifications the AST is generated using JastAdd's interface to Beaver parser generator and JFlex scanner generator. This AST is then extended with attribute definitions from both Java 1.4 and FeatureJ .jrag specifications. The final output is an attributed AST stored in terms of AST node classes in the folder AST and combined parser and scanners stored in parser and scanner folders respectively. Recall that Java 1.5 frontend extends Java 1.4 frontend. The FeatureJ parser and scanner specifications are not required to be included in Java1.5Frontend folder. Only the .jrag files that take care of new language constructs in Java 1.5 from the point of view of feature containments and name/type analysis and error checking are kept in

the FeatureJ folder of Java1.5Frontend folder. The rest of the structure in Java1.5Frontend folder is same as that of the Java1.4Frontend folder.

4.4 Summary

In this chapter, we elaborated how FeatureJ implements feature domain entities atop JastAdd Java 1.4 and 1.5 compilers. We used three extensibility techniques common in JastAdd, namely extending Java AST with typed representation, extending name/type analysis and error checking, and using AST rewrites for transforming extended syntax to Java equivalent source. Error checking and reporting is arranged so that variant-specific and productline-specific errors are conveyed separately for the variant types defined in a FeatureJ program. Combined with an architecture that overcomes Java class-loading idiosyncrasies and enables both compile-time and run-time composition of variant types, we get an easily extensible implementation of first-class features.

Chapter 5

rbFeatures

> Because of the Turing completeness theory, everything one Turing-complete language can do can theoretically be done by another Turing-complete language, but at a different cost. [...] Instead of emphasizing the what, I want to emphasize the how part: how we feel while programming.
>
> YUKIHIRO MATSUMOTO
> *in Philosophy of Ruby - A Conversation with Yukihiro Matsumoto*

In this chapter we elaborate how requirements of first-class features are implemented atop Ruby using Ruby's own syntax and semantics to obtain rbFeatures[1]. We first describe Ruby's extensibility mechanisms followed by rbFeatures implementation specifics including its syntax and semantics and its overall architecture.[2]

5.1 Ruby - Dynamic Extensible Host Language

Ruby is a dynamic programming language. Its creator Yukihiro "Matz" Matsumoto included language features from his favorite languages such as Perl,

[1] Sebastian Günther, an esteemed colleague of the author of this dissertation, implemented rbFeatures in collaboration with the author. The author thanks his colleague for a fruitful collaboration and appreciates many enjoyable hours of discussion that led to the implementation of first-class features in Ruby.

[2] This chapter shares material with the FOSD'09 paper *'Feature-oriented programming with Ruby'* [Günther and Sunkle 2009], its *extended Journal paper version* [Günther and Sunkle 2011] and the FOSD'10 paper *'Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming'* [Günther and Sunkle 2010].

Smalltalk, Eiffel, Ada, and Lisp with focus on balancing functional programming with imperative programming. Matz started developing Ruby on February 24, 1993 [Stewart 2001]. It was released publicly first in 1995. Ruby is open source, free to use, copy, modify, and distribute. By 2006, Ruby achieved mass acceptance.

Ruby is based on the *principle of least surprise*[3], which means that it is very easy to express programmers' intent with Ruby. It reduces a programmer's efforts in programming. It is designed to be *human-oriented* [Stewart 2001], i.e., programmers concentrate on the problem to be solved rather than having to write boilerplate code to get started. Ruby picks up the dynamic scripting features of python. At the same time, it enables programmers to do a certain task in more than one way, a feature which it borrows from Perl [Venners 2003a]. Most notable features in Ruby include the ability to change interface at run-time by adding methods and variables to objects at run-time, mixins which enable sharing methods between classes outside a single inheritance structure [Venners 2003c], blocks which are anonymous functions, and closures which are blocks of code that can be used as value, passed as parameters, executed on demand, and are able to refer to variables from the context in which they were created [Venners 2003b].

In the following, we focus on those characteristics of Ruby that we use in implementing rbFeatures. Unlike the implementation of FeatureJ using JastAdd, rbFeatures is neither implemented using an extensible compiler technology in Ruby nor uses separate lexer and parser components. Instead, we make use of the syntax and semantics of various Ruby language entities toward implementing a language extension. In the following, we explain the core Ruby language entities. These entities play a substantial role in representing feature domain entities at the language level which we explain in latter sections.

5.1.1 Core Language Entities in Ruby

The core language entities in Ruby are classes, methods, modules, and blocks. All Ruby data comprises of objects that are instances of some class [Matsumoto 2001]. Instances of classes are created with the `new` keyword. A method in Ruby is a named operation and contains the code that a specific class provides to perform the operation. Two kinds of methods exist in Ruby namely, class methods and instance methods. Class methods are methods that belong to class and must be called with respect to the class in which they are defined. Instance methods on the other hand are methods that must be called using an instance of a class, i.e., with respect to an object of the class in which the instance methods are defined.

[3]Ruby's creator Yukihiro Matsumoto says that he didn't particularly refer to this principle, rather it was attributed to him. However, he did create Ruby with the goal of minimizing the programming effort [Venners 2003a].

A module is similar to a class except that it does not have any superclass and cannot be instantiated. A module contains methods and constants which can be added to a class as well as an object. Ruby provides two ways to express inheritance relationships. The first is the standard inheritance relationship whereby a class extends or subclasses another class. The second is the *mixin* relationship which is a specialized implementation of multiple inheritance in which only the interface portion is inherited [Fulton 2006]. Unlike Java in which multiple inheritance is implemented indirectly using interfaces, the *mixin* relationship is realized in Ruby using a module as a mixin. A class can `include` a module and thereby its instance methods. A module's methods can be mixed-in with a class as both class methods and instance methods. Modules provide a secure namespace mechanism for classes at the same time enabling sharing of functionality and constants with other modules and classes.

Among its many ancestors, Ruby is said to inherit various aspects of object orientation from Smalltalk, whereas its functional programming capabilities have been taken from Lisp [Baird 2007]. These are based mainly on the concepts of blocks and procs. A block is an anonymous function. A block can be converted to and manipulated like an object using the Proc class. Such objectified blocks are referred to as *procs*. Although initially implemented as a way of providing loop abstraction, i.e., letting a programmer define his own way of iterating over any set of items, blocks have been used also as closures. A closure is a block that can be passed to a method to customize the behavior of that method [Venners 2003b].

In a preceding chapter, we discussed the concept of *first-class*ness of certain entities in a given language (cf. Section 3.4.1). In Ruby, classes and objects are first-class entities. Ruby's most distinctive characteristic is that "*Everything in Ruby is an object*". The fact that all language entities in Ruby can be governed in terms of objects of some class can be made clear by the Ruby object model illustrated in Figure 5.1. The classes shown in Figure 5.1 are the foundation of Ruby. The `BasicObject` class defines methods to create objects. It is extended by the `Object` class from which all classes and modules are inherited in Ruby. While `Object` is the parent class of all classes in Ruby, its core functionality is implemented in the Kernel module which it mixes-in and thus makes available to all classes. The Module class provides various reflection and metaprogramming facilities as we will discuss in the next section. Figure 5.1 also shows that the class Class which represent a Ruby class is a subclass of the Module class which is itself a subclass of the Object class. The two expression namely, `Class.is_a? Object` and `Object.is_a? Class`, are evaluated as `true` because of this hierarchy of core classes in Ruby. The fact that everything in Ruby is a object is of primary importance in the way in which we represent various feature domain entities in Ruby as we will discuss in the next section.

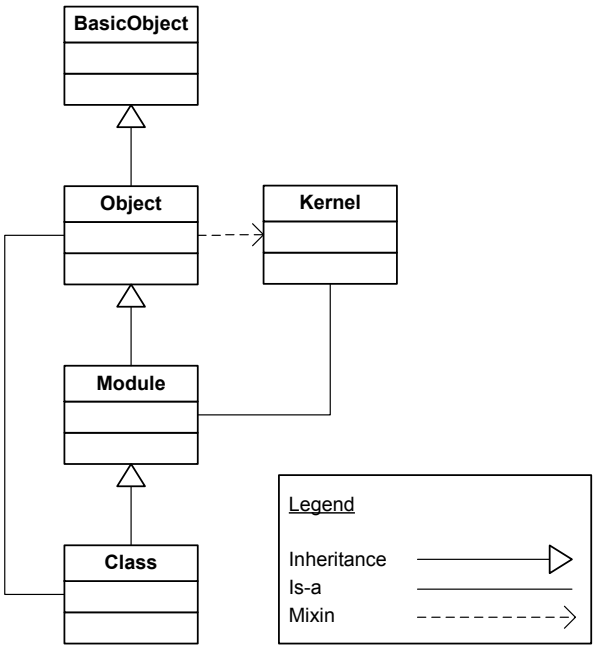

Figure 5.1: Ruby Object Model

5.1.2 Ruby Mechanisms for First-class Features

In this section, we review mechanisms available in Ruby that can be used toward implementing functionality of various feature domain entities with first-class status. We divide these into three kinds namely, basic mechanisms that use regular Ruby expressions but differ from other mainstream languages such as Java, reflection and metaprogramming mechanisms that can be used to address and query properties of entities in a Ruby program as well as manipulate them, and string manipulation mechanisms that enable selecting and modifying arbitrary pieces of code by treating parts of program as a string and manipulating this string representation. In the following, we discuss each of these mechanisms in turn.

Basic Ruby Mechanisms

Basic Ruby mechanisms consist of using a module as mixin and namespace mechanism, using callable objects including procs and method objects, and class reopening. We elaborate each in the following:

Modules as mixins and namespace mechanism Modules provide a mech-

anism of grouping together methods, classes, and constants. A module cannot be instantiated but when mixed-in with a class, its methods become instance methods of the class. This is illustrated in Listing 5.1. The text after # => indicates the output. The `include` statement only

Listing 5.1: Mixin Functionality with Modules [Thomas and Hunt 2000]

```
module Debug
  def whoAmI?
    "#{self.type.name} (\##{self.id}): #{self.to_s}"
  end
end
class Person
  include Debug
  # ...
end
class Thing
  include Debug
  # ...
end
person = Person.new("Bob")
thing = Thing.new("Chair")
person.whoAmI?   # => "Person (#537766170): Bob"
thing.whoAmI?      # => "Thing (#537765860): Chair"
```

makes reference to a named module. In order to `include` a module from another file, `require` statement can be used. If the methods of modules are needed to be added as class methods instead of as instance methods, the `extend` statement can be used instead of an `include` statement.

Another usage for modules is to act as a sandbox in which methods and constants that don't naturally form a class can be put together. For instance, module `Trig` defined in Listing 5.2 collects together methods and constants related to trigonometry.

Listing 5.2: Module as Namespace defined in File trig.rb

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
   # ..
  end
  def Trig.cos(x)
   # ..
  end
end
```

A module thus provides a namespace for these methods and constants. The trigonometry methods can be used as shown in Listing 5.3.

Listing 5.3: Using a Trigonometry Method from Module Trig

```
require "trig"
y = Trig.sin(Trig::PI/4)
```

Callable objects In Ruby, objectified methods as well as procs which are objectified blocks are treated as callable objects [Flanagan and Matsumoto 2008]. As discussed earlier, blocks of code can be objectified using the Proc class. Listing 5.4 shows that any block of code that is passed to the method procFrom() will be returned as an object which can then be called using the `call` method of the Proc class.

Listing 5.4: Converting a Block of Code to an Object

```
def procFrom
  Proc.new
end
aProc = procFrom { "Hello" }
aProc.call     # => "Hello"
```

Similarly, two forms of objectified methods are supported in Ruby. An instance of the Method class represents a method bound to an object. The Object class defines a method called `method` which returns an object of Method class that represents the method bound to an object. This is shown in Listing 5.5. In it, `:greeting` is a symbolic name. It is used to refer to the greeting method bound to the object obj of Person class. Unbound method on the other hand are methods objects that are not

Listing 5.5: Bound Method Objects

```
class Person
 def greeting
  "Hello"
 end
end
obj = Person.new
greeting = obj.method(:greeting)
greeting.call # => "Hello"
```

associated with a particular object. These are created using the method `instance_method` of the Module class. To call an unbound method, it

must be bound. This is shown in Listing 5.6 where `area_unbound` represents an unbound method object which is bound to the object s of class Square before calling it. Callable objects enable addressing blocks

Listing 5.6: Unbound Method Objects

```
class Square
 def area
  @side * @side
 end
...
end
area_unbound = Square.instance_method(:area)
s = Square.new(12)
area = area_unbound.bind(s)
area.call    # => 144
```

of code and both bound and unbound methods. With callable objects functionality can be added to an object or a method can be called in a free context.

Class reopening In Ruby, classes are available for modification at any stage of program execution. This applies to user classes as well as system classes. A class can be reopened to add a method or to redefine a method. Like everything else, there are many ways to add and redefine a method, e.g. using Ruby metaprogramming mechanisms, but class reopening provides a pre-runtime option to do the same. Listing 5.7 shows that the `Person` class is defined without any methods. Then method `greeting` is added to it and called. It is then redefined and called again with different result.

The basic Ruby mechanisms discussed so far already show how some functionality can be *grouped and added* (using modules), *objectified* (using objectified blocks and methods), and *modified* (using class reopening). In the next section, we elaborate the reflection and metaprogramming mechanisms in Ruby that go beyond basic mechanisms in terms of the capability to address and manipulate functionality.

Reflection and Metaprogramming in Ruby

Ruby provides comprehensive support for reflection and metaprogramming. In Ruby it is possible to use reflection to query properties of various language entities. For instance, it is possible to group methods for objects by their visibility and type, query about global and instance variables, and constants in a Ruby program. Table 5.1 enlists various reflection methods in Ruby. It shows

Listing 5.7: Adding and Redefining Methods by Reopening a Class

```
class Person
end

class Person
 def greeting
   "Hello"
 end
end
obj = Person.new
obj.greeting # => "Hello"

class Person
  def greeting
     "Bye!"
  end
end
obj.greeting # => "Bye!"
```

which properties of modules, classes, methods, and procs can be queried using various methods of Object, Class, Module, and Kernel which we described in the Ruby object model in Figure 5.1 on page 102.

Information can be obtained regarding most properties except the body internals. ObjectSpace module enables traversing all the living objects. The `Kernel#local_variables` returns the names of the local variables. While reflection methods in Ruby enable obtaining information, in order to modify functionality of various language entities we have to turn to metaprogramming.

Metaprogramming in Ruby is defined in [Perrotta 2010] as "*writing code that writes code*", more specifically, "*writing code that manipulates language constructs at run-time*". Table 5.2 shows methods of classes BasicObject, Class, and Module that can be used to modify variables, methods, and arbitrary blocks of code.

Ruby metaprogramming provides evaluation capabilities along with modification capabilities for various language constructs. The `Module#class_eval` evaluates a block of code in the context of an existing class. Listing 5.8 shows how `class_eval` can be used to add method `greeting` to the Person Class. It also shows how class reopening can be carried out dynamically instead of at pre-runtime as it was done in Listing 5.7.

Further support for capturing events in the object model is provided by Ruby in terms of *hook methods*. Hook methods are instance methods in Class and Module classes that can be overridden by a developer as required.

Listing 5.9 shows how `Class#inherited`() is overridden in String class so that whenever the String class is inherited a message is printed. In gen-

Object	**Object Property**	**Applicable methods**
Module/Class	*Name*	ObjectSpace#each_object, Class#superclass, Module#class, Module#nesting, Module#ancestors, Module#included_modules
	Methods	Object#protected_methods, Object#public_methods, Object#private_methods, Module#public_instance_methods, Module#private_instance_methods, Module#protected_instance_methods
	Class variables	Module#class_variables
	Instance variables	Object#instance_variables
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-
Method	*Name*	(via Module/Class methods)
	Parameters	-
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-
Procs	*Name*	-
	Parameters	-
	Body (Variables/Procs)	(Kernel#local_variables)
	Body (Other)	-

Table 5.1: Reflection Methods in Ruby

Object	**Method**	**Explanation**
Variables	Module#const_set, Module#remove_const	Set and remove constants
	Module#attr, Module#attr_reader, Module#attr_writer, Module#attr_accessor	Provide read, write or combined access to instance variables
	Object#instance_variable_set, Object#remove_instance_variable	Set and remove instance variables
	Module#class_variable_set, Module#remove_class_variable	Set and remove class variables
Methods	Module#alias_method	Copies method body to a newly named method
	Module#define_method	Defines new methods
	Module#undef_method	Prevents object to respond to calls of the method
	Module#remove_method	Deletes the method from module/object
Arbitrary	BasicObject#instance_eval	Evaluates string or block (defines class methods)
	BasicObject#instance_exec	Evaluates block with additional parameters (defines class methods)
	Module#module_eval, Module#class_eval	Evaluates string or block (defines instance methods)

Table 5.2: Built-in Metaprogramming Methods in Ruby

Listing 5.8: Using `class_eval` to Add a Method to a Class

```
class Person
end
def add_method_to(a_class)
end
def add_method_to(a_class)
  a_class.class_eval do
    def greeting; 'Hello'; end
  end
end
add_method_to Person
puts Person.new.greeting   # => "Hello"
```

Listing 5.9: Hook Methods in Ruby [Perrotta 2010]

```
class String
 def self.inherited(subclass)
 puts "#{self} was inherited by #{subclass}"
 end
end
class MyString < String;
end
# => String was inherited by MyString
```

eral, any Ruby code may be evaluated when a hook method is called. Like `Class#inherited`() method, Ruby provides `Module#included`() which can be used to track when a module is being included and `Module#extend_object`() which can be used to execute Ruby code when a module extends an object. Other important hook methods include `method_added`(), `method_removed`(), and `method_undefined`() methods in Module which can be used to execute method-related events [Perrotta 2010].

While the basic Ruby mechanisms provide pre-runtime facilities to modify functionality of various language constructs, using reflection and metaprogramming in conjunction enables modifying functionality and taking event-based actions at run-time. However, it is still not possible to modify internals of Ruby's core objects when modification must happen in other ways than adding or overwriting language constructs. In the following we discuss string manipulation facilities in Ruby which can support modifications in terms of wider range of granularity.

String Manipulation in Ruby

It is possible to process the static structure of a Ruby program, i.e., its source code, prior to the execution. By evaluating parts of source code or the entire source code again, modifications at run-time can be effectively addressed.

With such holistic manipulations, run-time modifications that take place thorough user interactions can be handled. E.g. a Ruby library called *Ruby2Ruby* can be used to transform arbitrary blocks of Ruby code to corresponding string representation. Such string representations can be processed, e.g, using regular expressions, altered and then re-evaluated by using various evaluation methods such as `class_eval` shown in Listing 5.8.

Utilizing Ruby Mechanisms for First-class Features

In the preceding sections, we reviewed various Ruby mechanisms that can be used to modify parts of a Ruby program and thereby alter its functionality. While basic Ruby mechanisms are capable of making modifications at pre-runtime, reflection and metaprogramming facilities can be used to affect the modifications at run-time. Modifications of arbitrary granularity are possible with string manipulation capabilities of Ruby. We use these mechanisms in conjunction to represent various feature domain entities. In the next section, we describe first how we represent them at language level and then elaborate the details of how the functionality of feature domain entities is implemented.

5.2 rbFeatures Language Internals

In order to describe the rbFeatures syntax and semantics for various feature domain entities, we make use of the Graph Product Line (GPL) [Lopez-Herrejon and Batory 2001]. Figure 5.2 shows a feature model of the GPL. In the following, we describe how SPLs representing the feature model, features, and product variants representing the product/variant models are represented at the syntactic level in rbFeatures.

5.2.1 Syntactic Extension

In this section, we describe rbFeatures syntax in terms of various keywords that represent various feature domain entities. We elaborate the actual implementation of these keywords in terms of module, classes, and procs in the next section.

An SPL is represented in rbFeatures by the ProductLine keyword as shown in Listing 5.10. Ruby's syntactic malleability enables creating syntax that corresponds to natural language. For instance, the intent of adding a feature

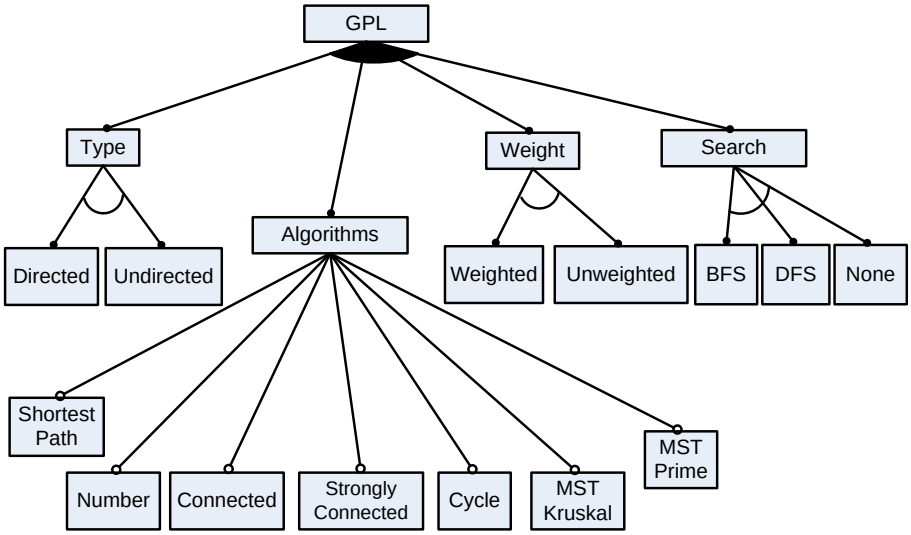

Figure 5.2: A Feature Model of Graph Product Line

Listing 5.10: ProductLine Definition for GPL in rbFeatures

```
GPL = ProductLine.configure do
 add_feature gpl_feature
 add_feature type_feature
 add_feature weight_feature
 ...
end
```

is indicated by `add_feature`. Listing 5.11 shows how a Feature is defined in rbFeatures. The name of a Feature is indicated by `name`. That the Feature GPL is the root feature is indicated by `root` keyword.

Listing 5.11: Feature Definition for Root Feature GPL in rbFeatures

```
gpl_feature = Feature.configure do
 name :GPL root
 subfeatures :Type, :Weight, :Search, :Algorithms
 requires :GPL =>
  "more :Type, :Weight, :Search, :Algorithms"
end
```

The children of Feature are indicated by keyword `subfeatures`. The keyword `requires` along with the keyword `more` indicates that it is required to select

one or more of Type, Weight, Search, and Algorithms Feature entities when the Feature GPL is activated. There are no constraints in the feature model of the GPL being referred to (a constraint can be indicated in a manner similar to FeatureJ, i.e., using special operators <–> and >–<).

Listing 5.12: Declaring Feature Entities inside Ruby Code in rbFeatures

```
class Weighted
 is Feature
end
...
class Undirected
 is Feature
end
...
```

The underlying feature model is mirrored in the definitions of the rest of the Feature entities indicated in Listing 5.10 by type_feature and weight_feature etc. which are represented similarly as in Listing 5.11.

Listing 5.13: Containment for Feature Directed and Feature Undirected inside Class Graph

```
class Graph
 def initialize (gtype)
  Directed.code { def directed?; true; end if gtype==1 }
  Undirected.code { def directed?; false; end }
  ...
end
```

Listing 5.14: Feature Weighted inside Class Edge

```
class Edge
 Weighted.code {attr_accessor :weight}
 def intialize (params)
  Weighted.code { @ weight = params.delete :weight }
  params.delete :weight if params.include? :weight
  ...
 end
end
```

Individual Feature entities in the code are indicated by the keyword `Feature` as shown in Listing 5.12. The fact that the class Weighted is a Feature is indicated using the `is` keyword. The optionality of a Feature can be indicated using `mandatory` variable and setting it to false inside the definition of a Feature.

Listing 5.15: ProductVariant SimpleGraph of ProductLine GPL

```
1 ProductVariant.configure
2   :name => "SimpleGraph", :pl => GPL do
3   activate_features :Directed, :Weighted, :DFS,
4     :Strongly_Connected
5 end
```

The code fragments that belong to a specific Feature are indicated using the keyword code and a pair of curly braces inside the regular Ruby code. Listing 5.13 shows the code in Graph class that belongs to Feature entities Directed and Undirected. Listing 5.14 shows the code in Edge class that belongs to Feature Weighted.

Listing 5.16: Defining Multiple Variants of ProductLine GPL

```
1 ProductVariant.configure
2   :name => "ShortestPathVariant", :pl => GPL do
3   activate_features :Directed, :Weighted, :ShortestPath,
4 end
5 ProductVariant.configure
6   :name => "DFSVariant", :pl => GPL do
7   activate_features :Directed, :Weighted, :DFS,
8 end
```

A variant of the GPL in rbFeatures is indicated by the keyword ProductVariant. Listing 5.15 shows the definition for SimpleGraph ProductVariant. The fact that it is a variant of the GPL is indicated by `:pl => GPL`. The intent of configuring an SPL by selecting specific features is indicated by two keywords namely, `configure` and `activate_feature`. We use the keyword `activate_feature` instead of `select_feature` with the rationale that owing to the dynamic nature of Ruby, features can be assumed to be activated or deactivated dynamically instead of being selected statically.

More than one variant of an SPL can be defined at the same time as shown in Listing 5.16. Two ProductVariant entities namely, ShortestPathVariant and DFSVariant, that indicate variants of the GPL ProductLine. Listing 5.17 shows how ShortestPathVariant- and DFSVariant-specific graphs are initialized. These graph objects provide the functionality that is available according to the underlying ProductVariant.

5.2.2 Feature Domain Entities in rbFeatures

Having described the syntax adopted by rbFeatures to represent various feature domain entities, we now elaborate how these entities are implemented along

Listing 5.17: Initializing ProductVariants of the GPL

```
GPL.variant("ShortestPathVariant").instantiate!
ShortestPathVariant.class_eval do
 graph=Graph.new
 1.upto(6) { |n| graph + node(n) }
 graph + edge(1 => 2, :weight => 1)
 graph + edge(1 => 3, :weight => 2)
 graph + edge(2 => 5, :weight => 4)
 graph + edge(2 => 4, :weight => 2)
 graph + edge(4 => 6, :weight => 12)
 graph + edge(3 => 6, :weight => 22)
 ShortestPathGraph= graph
end

GPL.variant("DFSVariant").instantiate!
DFSVariant.class_eval do
 graph=Graph.new
 1.upto(6) { |n| graph + node(n) }
 graph + edge(1 => 2, :weight => 1)
 graph + edge(1 => 3, :weight => 2)
 graph + edge(2 => 4, :weight => 3)
 graph + edge(3 => 5, :weight => 4)
 graph + edge(4 => 5, :weight => 4)
 DFSGraph= graph
end
```

with various keywords that we mentioned in the preceding section. We begin by describing the internals of three main entities in rbFeatures namely, Feature, ProductLine, and ProductVariant.

The Feature Module

We implement Feature as a module, which is a combination of Core, MethodAddedHook, and other modules. The most important modules and methods in Feature module are shown in Figure 5.3. Each supports part of the functionality for representing classes in terms features when the Feature module is mixed-in with the classes such as Weighted, Directed, and Undirected etc. as shown in Listing 5.12 on page 111. Below, we describe each of these modules:

Core This module provides the functionality to activate or de-activate a Feature with activate() and deactivate() methods that keep track of the activation status of a Feature using active and temp_active instance variables. The method **code**() is the keyword code shown in Listings 5.13 and 5.14. The block of code surrounded by code{...} constitutes a feature containment. A feature containment is objectified by converting it to a proc.

The objectified feature containment is used in evaluating Feature-specific code when activating a Feature.

MethodAddedHook This module provides definitions for the hook methods `method_defined`() and `singleton_method_defined`(). When feature containments consist of a method, `method_defined`() is called. When the Feature that defines the containment on a block of code using .code{...} has not been activated but the contained method is called, then the method body is replaced with an error message which reports the details about Feature which caused the error. For instance, method directed() was called on an instance of class Graph but Feature Directed was not activated, then the error message reports this situation. The error message itself is constructed dynamically. The `apply_visibility` method retains the visibility of the original method in case it is replaced by another method.

Other modules inside the Feature module similarly define hooks for other entities such as instance variables and provide syntactic sugar for multiple Feature containments by overriding operators such as +.

The ProductLine Class

An SPL is represented in rbFeatures by an instance of ProductLine class. The ProductLine class stores the name of the SPL it is referring to, the complete application code as a string, and a list of the variants that are declared in the program. This list is updated in a given instance of a ProductLine class if a ProductVariant is created based on the SPL it refers to which is added immediately to the list using the `add_variant`() method. The structure of the ProductLine class is shown in Figure 5.4.

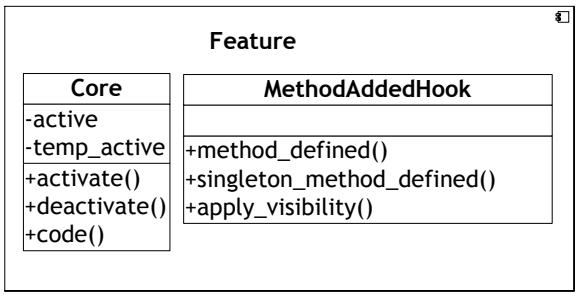

Figure 5.3: Structure of the Feature Module in rbFeatures

The `name`(), `add_feature`(), and `remove_feature`() methods are used to set the name of the SPL referred to by an instance of a `ProductLine` class, and add features to or remove features from the SPL definition itself. The

`activate_features_in_variants`() and `deactivate_features_in_variants`() methods oversee activation and de-activation of features respectively, from instances of ProductVariant class that represent variants of this product line. The `valid`() method is applied after the required features have been added to an instance of a ProductLine class that they represent a valid feature model, e.g., that it is a tree rather than a cyclic graph. The checks are similar to the checks applied to a productline type in FeatureJ.

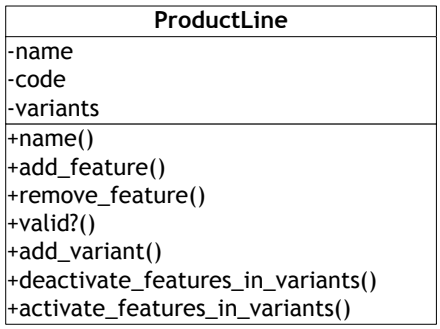

Figure 5.4: Structure of the ProductLine Class in rbFeatures

The ProductVariant Class

A product/variant entity is represented in rbFeatures by an instance of ProductVariant class. The ProductVariant stores the name of the variant entity it represents and the name of the ProductLine class to which it belongs. It contains the methods that represent keywords which we have seen earlier in Listings 5.10 to 5.11 on page 110.

The ProductVariant class consists of method instantiate() which returns an instance of this class. The method `activate_features` is used to initially choose features that make up a variant entity. On the other hand, method `activate_features_in_instance` is used to activate features in an already instantiated variant entity, i.e., in an instance of the ProductVariant class. The methods `deactivate_features` and `deactivate_features_in_instance` are similarly used to deactivate features.

The variable `feature_tree` stores the symbols of all activated features. This variable as well as methods that represent `one, any, more, all` keywords are used in validating the product/variant model that an instance of ProductVariant class represents.

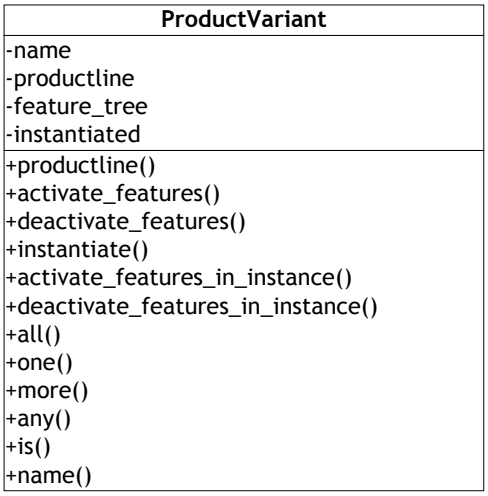

Figure 5.5: Structure of the ProductVariant class in rbFeatures

Other Important Modules and Classes in rbFeatures

A module called FeatureResolver and a class called FeatureModel are used internally in rbFeatures. The FeatureResolver module oversees the initial evaluation of Feature related code and later modification at run-time. Its structure is shown in Figure 5.6.

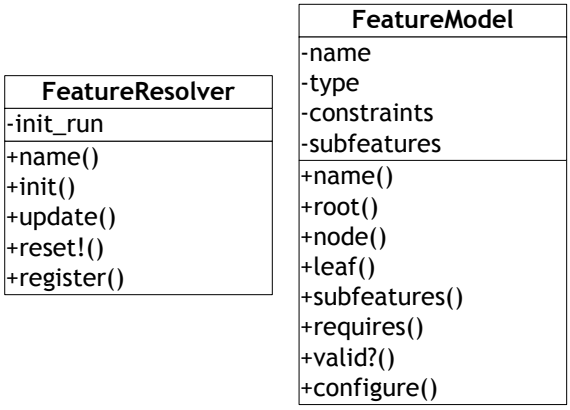

Figure 5.6: Other Important Modules and Classes in rbFeatures

The `init`() method in FeatureResolver module takes the entire program as an

objectified proc. The entire program is evaluated once so that each activated Feature per ProductVariant registers itself to the FeatureResolver. In combination with the functionality provided in Feature module, any modifications to the configuration of a ProductVariant instance can be recognized at run-time. The current activation status of a Feature per ProductVariant is kept updated with the `update`() method. With this functionality, FeatureResolver is able to keep track of features that are activated or de-activated at run-time. When the activation status of a Feature changes, FeatureResolver is used to re-evaluate the application code including validation of ProductVariant with respect to its parent ProductLine.

Like the FeatureResolver module, FeatureModel is used internally. Listings 5.10, 5.15, and 5.16 use the keyword `configure`, which is actually the method `configure` in FeatureModel class which returns an instance of feature class that is mixed-in with Feature module. Both the ProductLine and ProductVariant classes use an instance of FeatureModel class that represents the feature model of an SPL and a product/variant model respectively. The Feature module and the ProductLine, ProductVariant, and FeatureModel classes are interconnected as shown in Figure 5.7.

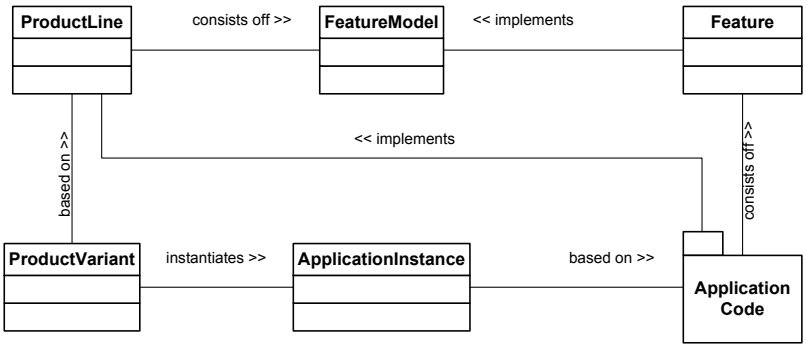

Figure 5.7: Relations Between Feature Domain Entities in rbFeatures

5.2.3 Testing in rbFeatures

To validate an instance of ProductVariant class representing a variant, the product/variant model implicitly stored in the variable `feature_tree` is used. The variable `feature_tree` stores the symbols of all activated features. When a ProductVariant instance is created, its feature tree is traversed using `feature_tree`, accessing each `Feature` in turn, executing various methods indicating the relations. For instance the line `"more :Type, :Weight, :Search, :Algorithms"` shown earlier in 5.11 on page 110 is in fact a method call for method more() in

the ProductVariant class shown in Figure 5.4 on page 115. Entire feature tree stored in `feature_tree` is evaluated to check whether each relation is satisfied.

The testing of application code itself takes place using RSpec which is a Ruby API for behavior/test-drivend development in Ruby [Chelimsky et al. 2010]. RSpec is based on the concept that behavior/tests drive the implementation [RSpec 2010]. New tests are added to the specification and the application is incrementally developed. The same style is used when developing an rbFeatures SPL. Note that while validation of an instance of ProductVariant class takes place by ProductVariant class's specification, individual application classes such as Weight and Directed etc. must be tested as they are coded. This is in contrast to static error checking provided by FeatureJ. In rbFeatures tests for the correct application of various semantic mechanisms take place inside these classes. These include tests:

- to cover feature containments (in `code` surrounded blocks) for different granularities.
- to check that methods retain their visibility after modification.
- to check the correct execution of feature activation using evaluation of containments.
- to check that application is correctly working in the specified context and modifications to ProductVariant class instances stay in the ProductVariant-specific scope.

Contrary to error checking support in FeatureJ, the developer needs to provide a complete set of tests for the whole application when coding SPLs in rbFeatures. We find the correct way to do this is to structure the test suite according to features to be tested and then test those configurations that are to be instantiated as ProductVariant instances. Errors in these tests are taken as indication for incomplete feature containments.

5.3 Architecture and Implementation Statistics

In this section, we describe the architecture that enables more than one ProductVariant instances possible in rbFeatures and provides support for run-time adaptation of application class instances.

5.3.1 Architecture for rbFeatures

Recall from our discussion of variant generation architecture in FeatureJ on page 90, that generating versions of more than one variant-specific application classes constitutes providing support for separate namespaces where variant-specific application classes live and are loaded from. Unlike Java, Ruby has no concept of loading classes using class loaders or a folder hierarchy that

mimics the package structure of a Java application. Owing to the dynamic and interpretive nature of Ruby, instead of creating separate namespace per variant type as in FeatureJ, we create separate scopes per ProductVariant in rbFeatures. This is illustrated in Figure 5.8.

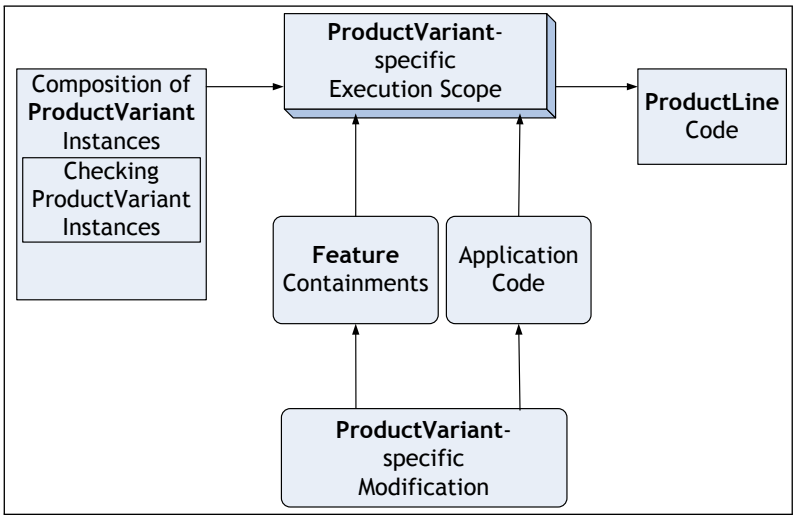

Figure 5.8: rbFeatures Architecture for First-class Features

Like the FeatureJ architecture for composing variants shown in Figure 4.11 on page 91, the arrows in Figure 5.8 indicate general flow of control within various entities in rbFeatures. Initially, the complete application is stored as a string object. Consider a statement that instantiates a ProductVariant as shown in the first line `GPL.variant("ShortestPathVariant").instantiate!` in Listing 5.17. When method `instantiate` is called following steps are triggered:

1. Check whether the features activated in ProductVariant instance are valid based on the ProductLine.
2. Compose a string consisting of a module and the SPL code. The module uses the configured name of the variant and serves as a namespace.
3. Add the string to the FeatureResolver.
4. Initialize the application by evaluating the string.

Note that while run-time composition of ProductVariant instances is default in rbFeatures, separated scopes also enable run-time adaptation as feature activation status of Feature mixed-in classes is synchronized. Due to the fact that each ProductVariant instance and corresponding application classes reside in an exclusive scope, modifications such as feature addition/removal affects each ProductVariant instance individually.

This is particularly effective when application class objects maintain state. For instance, in the GPL, individual graph have states such as edge directions, weight assignment of edges etc. Unlike FeatureJ, it is possible to alter the schema of an application class (i.e., add or remove a method of the class) and the modification is reflected immediately to an instance of this class. A detailed example of modifications in GPL is provided in the next chapter in the review of rbFeatures case studies (cf. Section 6.2.1 on page 144).

5.3.2 Implementation Statistics

Table 5.3 shows the implementation statistics for rbFeatures which is implemented atop Ruby version 1.9. Note that the difference between code sizes in Tables 4.1 and 5.3 is due to the fact that rbFeatures is implemented atop Ruby using Ruby's own syntactic and metaprogramming facilities whereas FeatureJ implementation atop JastAdd consists of the implementations of Java 1.4 and 1.5 compilers which are extended with corresponding syntactic (lexer and parser) and semantic (AST representation) components. It also demonstrates facility provided by malleable syntax and metaprogramming techniques in Ruby toward implementing an internal DSL.

Ruby Language Extension	Syntax SLOC	Semantics SLOC	Testing SLOC
rbFeatures	47	768	1964

Table 5.3: rbFeatures Implementation

It is possible that if we had chosen to implement rbFeatures by extending the base C implementation, we would have faced similar complexity and needed to take care of larger code sizes as in FeatureJ implementation.

5.4 Summary

In this chapter we elaborated how we implemented first-class features in Ruby. As a dynamic programming language, Ruby offers different execution model from Java with some default language features such as open classes and mixins which are used in the implementation of feature domain entities. The string manipulation and metaprogramming support in Ruby enable equivalent treatment of code and data which results in a concise implementation of first-class features. Instead of composing AST, parser, and scanner specifications like in FeatureJ with JastAdd, rbFeatures exploits Ruby's malleable syntax. We also leverage the fact that Ruby has no compile-time and enable run-time adaptation of ProductVariant-specific application class instances using Ruby's ability to evaluate and execute code within a given context.

Chapter 6

Evaluation

> However, it isn't always necessary to choose between function and elegance. [...] What is necessary is to design the product so that newly added features do not eliminate useful capabilities, make good use of capabilities already present for other purposes, and can be ignored or deleted by people who don't want them.
>
> David Parnas
> *in Why software Jewels Are Rare*

In this chapter, we explain the implementation of four SPLs each in FeatureJ and rbFeatures and also show that both FeatureJ and rbFeatures conform to the requirements of first-class features. We present two case studies, namely Expression Product Line (EPL) and Graph Product Line (GPL) in both FeatureJ and rbFeatures, while the rest two case studies in each are different. We use alternate feature models for EPL and GPL in FeatureJ and rbFeatures. The other case studies demonstrate application of first-class features in the context of GUI-based SPLs such as Notepad Product Line (NPL) in FeatureJ and Calculator Product Line (CPL) in rbFeatures and finally, one large case study each, Berkeley DB Product Line (BDBPL) in FeatureJ and Twitter Application Product Line (TAP) in rbFeatures.[1]

[1] EPL and NPL in FeatureJ were presented in [Sunkle et al. 2009] and [Sunkle and Pukall 2010] and CPL and GPL in rbFeatures were presented in [Günther and Sunkle 2009] and [Günther and Sunkle 2010] respectively. An alternate feature model of GPL was used in [Sunkle et al. 2008b] and TAP is presented in the extended Journal paper version [Günther and Sunkle 2011].

6.1 FeatureJ

6.1.1 Case Studies

Table 6.1 shows each SPL, number of compilation units in each, number of features and SLOC. The number of features indicate all features including the root and intermediate features as well as the leaf features that actually contain code fragments in the application code. We also present the origins of the EPL and GPL as SPL problems. The EPL is coded in FeatureJ with the same feature model implicitly used in the case studies in [Lopez-Herrejon et al. 2005a] which demonstrate application of AspectJ, Hyper/J, AHEAD, Jiazzi, and Scala to implement the EPL. We use it to compare FeatureJ with these feature implementation techniques.

The feature model of GPL used in FeatureJ is different from one already presented in rbFeatures in Chapter 5. It is a mid-size SPL which we used during initial implementation of FeatureJ itself to test its functionality.

We demonstrate output of multiple notepad variants explained earlier with NPL in FeatureJ in Chapter 4. The NPL is midway between EPL and GPL in terms of code size, but provides opportunity to demonstrate FeatureJ syntax and semantics which we capitalized in Chapter 4. We also give an example of error checking in FeatureJ which we discussed earlier in Section 4.2.3 on page 84.

Finally, we explain the adoption of the implementation of BDBPL originally presented in [Kästner et al. 2007]. Although extremely large compared to other SPLs, the effort involved was eased due to its original implementation.

SPL	Compilation Units	Features	SLOC
EPL	8	10	169
NPL	4	17	500
GPL	8	15	1K
BDBPL	308	55	45K

Table 6.1: FeatureJ Case Studies

Expression Product Line

The EPL is based on the *expression problem*. The expression problem is a design problem concerned with extending data abstractions to support new data operations and representations [Cook 1991; Wadler 1998]. Discussions related to the expression problem consider type-safety issues when considered from programming language design perspective. Lopez-Herrejon [Lopez-Herrejon 2004] presented it from design and synthesis perspective. They later treated

it as a case-study where five feature modularization techniques were used to implement it as an SPL (i.e., as EPL) in [Lopez-Herrejon et al. 2005a]. This is the same SPL which we earlier referred to within the context of capabilities of feature modularization techniques (cf. Section 3.2 on page 31). The

	Print	Eval
Lit	lp	le
Add	ap	ae
Neg	np	ne

Figure 6.1: Expression Problem as Two-Dimensional Matrix

solution to the expression problem itself has been represented in terms of a two-dimensional matrix [Cook 1991; Wadler 1998], that arranges data types along the rows and data operations along the columns as shown in Figure 6.1. The combinations of data types with operation are treated as features in [Lopez-Herrejon et al. 2005a; Lopez-Herrejon 2004]. Lopez-Herrejon et al. do not present a feature model of the EPL, since the stress in the solution in AHEAD [Lopez-Herrejon 2004] and later in terms of other techniques [Lopez-Herrejon et al. 2005a] is on feature modularization rather than feature modeling. We convert the two dimensional matrix representation of the expression problem in Figure 6.1 to a feature model representing the EPL as illustrated in Figure 6.2. This feature model is implicit in the implementation of the EPL in the five feature modularization techniques compared in [Lopez-Herrejon et al. 2005a].

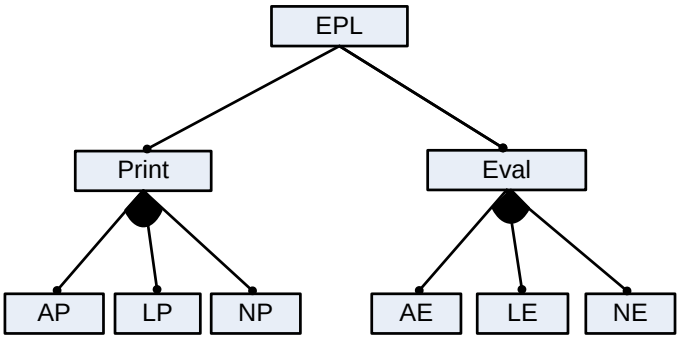

Figure 6.2: Expression Product Line in FeatureJ

The details of individual implementations of the EPL in each of AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD can be found with nearly complete source code in the extended version of [Lopez-Herrejon et al. 2005a] published as a technical report [Lopez-Herrejon et al. 2005b]. Similarly, the source code for the FeatureJ version of the EPL was first presented in [Sunkle et al. 2009].

Listing 6.1: Definition of Feature le in AspectJ

```
// LE.java
public aspect LE {
 public abstract int Exp.eval();
 public int Lit.eval() { return value; }
  pointcut LPRun(Test t):execution(public void Test.run())
   && target(t);
 void around(Test t) : LPRun(t) {
  proceed(t);
  System.out.println(t.ltree.eval());
 }
}
```

Listing 6.2: Definition of Feature le in Hyper/J

```
// Exp.java
package le;
interface Exp { int eval();}
// Lit.java
package le;
class Lit implements Exp {
 int value;  // stub lp
 Lit (int v) { } // req constructor
 public int eval() { return value; }
}
// Test.java
package le;
class Test {
 Lit ltree; // stub lp
 void run() {
  System.out.println(ltree.eval());
 }
}
```

Of the five feature modularization techniques detailed in [Lopez-Herrejon et al. 2005b], four build upon Java, i.e., they introduce new modular entities to Java namely, aspects, hyper-slices, atoms and units, and refinements in AspectJ, Hyper/J, Jiazzi, and AHEAD respectively. Here, we only bring forth the salient differences in terms of representation and composition of feature domain entities in these techniques and FeatureJ with respect to the EPL using source code excerpts from [Lopez-Herrejon et al. 2005b] and the FeatureJ EPL implementation.

The non-SPL version of the expression problem consists of Exp interface with print() and eval() methods and three classes Add, Neg, and Lit that implement

Listing 6.3: Definition of Feature le in AHEAD

```
// Exp.jak
refines interface Exp { int eval(); }
// Lit.jak
refines class Lit implements Exp {
 public int eval() { return value; }
}
// Test.jak
refines class Test {
 public void run() {
  Super().run();
  System.out.println(ltree.eval());
 }
}
```

Listing 6.4: Definition of Feature le in Jiazzi - I

```
// Exp.java
package le;
 public interface Exp extends lp.Exp { int eval(); }
// Lit.java
public class Lit extends lp.Lit implements fixed.Exp {
 public Lit(int n) { super(n); }
 public int eval() { return value; }
}
// Test.java
package le;
public class Test extends lp.Test{
 public Test() { super(); }
 public void run() {
  super.run();
  System.out.println("= " + ltree.eval());
 }
}
public class Test { public void run(); }
}
```

Exp. Another class Test is used to test the printing and evaluation of additive, negative, and integer expressions represented by Add, Neg, and Lit classes. We use the source related to definitions of features LE, LP, AE, and AP. Note that original source refers to both lowercase and uppercase names of these features which we keep as they are in the original source.

In AspectJ, feature le is represented in an additional source file LE.java in terms of an aspect LE as shown in Listing 6.1. The functionality of feature le is concerned with evaluating integer expressions. Listing 6.1 shows inter-

Listing 6.5: Definition of Feature le in Jiazzi - II

```
1  // le.unit
2  atom le {
3   import lp : lpS;
4   export le extends lp : leS;
5   import fixed extends le;
6  }
7  // leS.sig the signature of unit le
8  signature leS = l : lpS + {
9   package fixed;
10  public interface Exp { int eval(); }
11  public class Lit {
12   public Lit(int n);
13   public int eval();
14 }
```

Listing 6.6: Definition of productline EPL in FeatureJ

```
1 public class Launcher {
2  productline EPL {
3   features {
4    Print : more(lp, ap, np),
5    Eval  : more(le, ae, ne)
6   }
7   all(Print, Eval)
8  };
9 }
```

type declarations in Lines 3 and 4 that add eval() method to Exp and Lit. AspectJ pointcut and advice mechanism is used in Lines 5-10 in Listing 6.1 which execute additional code using around advice.

While in AspectJ, a separate source file containing an aspect represents a feature, each of Hyper/J, Jiazzi, and AHEAD represent feature le with in Exp, Lit, and Test classes as shown in Listings 6.2, 6.3, and 6.4.

In Hyper/J, feature le is represented in terms of a hyper-slice coded as package le as shown in Listing 6.2. While hyper-slices that introduce new classes and interface represented in regular Java packages, representation of features such as le that introduce methods and other entities ascribe extra semantics to Java packages [Lopez-Herrejon et al. 2005b].

AHEAD uses refinements to interface Exp, and classes Lit and Test classes to represent feature le. The `Super` keyword adds additional syntax and semantics to a method call in regular Java [Lopez-Herrejon et al. 2005b]. For instance, Super().run() calls the run() method of the super class before executing the

Listing 6.7: Definition of feature le in FeatureJ

```
1  //Exp.fjava
2  public interface Exp {
3   feature EPL le { public int eval(); }
4  }
5  //Lit.fjava
6  public class Lit implements Exp {
7   feature EPL lp, EPL le {
8    int value;
9    public Lit(int v) {
10    value = v;
11   }
12  }
13  feature EPL le {
14   public int eval() {
15    return value;
16   }
17  }
18 }
19 //Test.fjava
20 public class Test {
21  feature EPL lp, EPL le { Lit ltree; }
22  public Test() {
23   feature EPL lp, EPL le { ltree = new Lit(3); }
24  public void run() {
25   feature EPL le { System.out.println(ltree.eval()); }
26  }
27 }
```

code that follows in the run() method of Test class as shown in Listing 6.3. All the code fragments shown in Listing 6.3 are arranged in a folder on disk (named le). The .jak extension acknowledges additional syntax such as the `refines` keyword.

Jiazzi similarly represents feature le in Exp interface and Lit and Test classes. Additionally, it extends the semantics of Java packages so that functionality of feature le can be grouped together as in package le as shown in Listing 6.4. `fixed.exp` indicates version of Exp that contains all extensions in a common composition indicated feature le. Additionally two files namely le.unit and leS.sig need to be coded to indicate feature le in terms of atom le [Lopez-Herrejon et al. 2005b], as shown in Listing 6.5.

It can be observed in Listings 6.2-6.5 that each of the feature modularization techniques extend the semantics of regular Java entities such as packages and method calls. The fact that certain code fragments constitute a feature is implicitly indicated in terms of packages or folders named after the features.

Listing 6.8: Composition of Variant LitAdd in Hyper/J

```
// LitAdd.hs
hyperspace  LitAdd
 composable class LP.*; composable class LE.*;
 composable class AP.*; composable class AE.*;
// LitAdd.cm
package LP : Feature.LP package LE : Feature.LE
package AP : Feature.AP package AE : Feature.AE
// LitAdd.hm
hypermodule LitAdd
 hyperslices:
  Feature.LP, Feature.AP,
  Feature.LE, Feature.AE;
 relationships:
  mergeByName;
end hypermodule;
```

Listing 6.9: Composition of variant LitAdd in FeatureJ

```
public class Launcher  {
 variant EPL LitAdd {
  Print  = [lp and ap],
  Eval = [le and ae]
 };
}
```

In FeatureJ, the EPL is indicated by productline type EPL and feature le by feature type le. Listing 6.6 shows the productline type definition of EPL in FeatureJ. Listing 6.7 shows the FeatureJ implementation of feature le. Code fragments that constitute feature le are scattered in interface Exp and classes Lit and Test and are indicated in them by feature containments.

Note that in Listing 6.7, some code fragments belong to multiple features including feature le. The fact that a code fragment is part of a feature is indicated by a feature containment. The semantics of none of the regular Java entities is extended. The feature le is explicitly specified and furthermore, the fact that it is a feature of productline EPL is also indicated.

When composing feature le with other features thus represented to obtain variants such as LitAdd which represents additive integer expressions, each of AspectJ, Hyper/J, Jiazzi, AHEAD need to take into consideration additional meaning ascribed to regular Java entities. In [Lopez-Herrejon et al. 2005b], composition examples are given only for Hyper/J and Jiazzi, which we consider next.

Listing 6.8 shows that to compose a variant of EPL that is capable of adding

Listing 6.10: Composition of Variant lelp in Jiazzi

```
1  // lelp.unit
2  compound lelp {
3  export compLELP : leS;
4  bind package compLELP to compLELP@fixed; }
5  {
6  link unit lpInst : lp, leInst : le;
7  link package
8   leInst@le to *@fixed,
9   lpInst@lp to leInst@lp,
10  leInst@le to compLELP;
11 }
```

Listing 6.11: Composition of variant lelp in FeatureJ

```
1  public class Launcher  {
2   variant EPL lelp {
3    Print  = [lp],
4    Eval = [le]
5   };
6  }
```

integer expressions in Hyper/J. It is required to specify it as a hyperspace, packages must be ascribed to features, and then these feature packages are merged at the byte-code level. On the contrary, the same variant is described in FeatureJ as shown in Listing 6.9. The FeatureJ version of expressing a variant as in Listing 6.9 is concise and to the point. The SPL and its variant are clearly related and the developer is only concerned about selecting features that make up a variant.

Similarly, in Jiazzi, composing a variant of EPL that is capable of evaluating and printing integer expressions (i.e., composition of feature le with feature lp) requires indicating a compound lelp that binds packages declared earlier in Listing 6.4 (feature lp in Jiazzi is not shown here, which can be referred to in [Lopez-Herrejon et al. 2005b]). Furthermore, various units must be linked together as shown in Listing 6.10.

The same variant is represented in FeatureJ as shown in Listing 6.11. Only feature types lp and le need to be selected from EPL and the variant is indicated by variant type lelp.

Composition of variants of EPL in AspectJ and AHEAD is not shown in [Lopez-Herrejon et al. 2005b]. In AspectJ, another ordering aspect is created that indicates the precedence in which aspects indicating features need to be weaved. Similarly in AHEAD, an equation file must specify the order of composition of refinements representing features. The EPL case-study in FeatureJ showcases

the following characteristics as compared to other implementation techniques as follows:

1. While other feature implementation techniques do not explicate the structure of the EPL or its variants, FeatureJ does in terms of productline type EPL and variant types LitAdd and lelp.
2. In FeatureJ, semantics of feature composition is separated into productline, variant, and feature types unlike other feature implementation techniques in which regular Java entities are overloaded with additional semantics that must be taken into consideration when composing features.
3. Ordering is important particularly to AspectJ and AHEAD which use ordering aspect and equation file respectively for the same. In FeatureJ, ordering is not required. The developer only has to select features from an SPL that make up a variant.

The EPL is a very small SPL. The code for the expression problem in terms of Exp interface, and Lit, Neg, Add, and Test classes fits one page and the SLOC number of EPL in FeatureJ is only 165. The consideration of additional semantics of regular Java entities and lack of explicit representation of an SPL and its variants can become very complex when larger SPLs are considered. We now move on to the description of other case studies in FeatureJ with larger code base starting with the NPL.

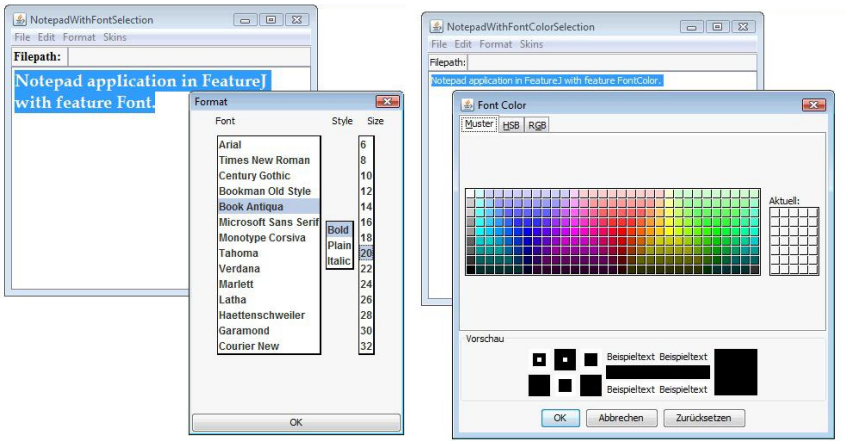

Figure 6.3: Two Notepad Variants

Notepad Product Line

We have used the NPL as the example SPL throughout Chapter 4 to demonstrate FeatureJ syntax and semantics. We refer to the feature model of the

NPL originally shown in Figure 4.7 on page 76. As specified earlier, it contains four compilation units and measures 500 SLOC. It also showcases the application of FeatureJ to GUI-based applications. We showed two NPL variant types namely, simpleNotepad and colorNotepad in Listings 4.16 on page 77 and 4.17 on page 78 respectively.

As shown in Listing 4.20, objects of the NPL application class NotepadGUI (i.e., one of the 4 compilation units that constitute the NPL), are obtained and run() method shown in Listing 4.21 is called which initiates Java GUI. The result is shown in Figure 6.3.

The NPL variant simpleNotepad is shown on the left in Figure 6.3. It provides the functionality to select font through the Font feature, which is a child feature of feature Format. The NPL variant shown on the right is colorNotepad variant. It provides the functionality to select the color of the text but not the ability to set the font type or size as in variant simpleNotepad.

Listing 6.12: An Unreachable Statement in NPL

```
1  public class Notepad extends JFrame {
2   feature notepadPL Font {
3    class FormatFont implements ActionListener {
4     public void actionPerformed(ActionEvent e) {
5      Mydialog md=new Mydialog(null);
6      return;
7      md.show();
8      }
9     }
10   }
11  ...
12  }
```

We also demonstrate error checking and reporting in FeatureJ with an example of the same from the NPL. Listing 6.12 an excerpt of the NPL in which a code fragment contained in feature Font contains an unreachable statement. Note that while variant simpleNotepad contains feature Font, variant colorNotepad does not. Recall from our discussion about FeatureJ error reporting that we distinguish between variant- and productline-specific errors and report errors for individual variants (cf. Section 4.2.3 on page 84). When a compilation-error such as the one shown in Line 6 of Listing 6.12 is present, FeatureJ reports the following error:

"*Unsafe Composition Error : In the variant simpleNotepad statement is unreachable in .\testNotepad\Notepad.fjava at line 471:.* "

Note that in the actual implementation of NPL the code fragment between Lines 2-10 in Listing 6.12 happens to be between Lines 466-474 and `md.show`↩ `()`; is at Line 471. The fact that due to the `return`; statement at Line 470,

the statement at Line 471 becomes unreachable and that this will affect only variant simpleNotepad is made clear in the error reported by FeatureJ.

Graph Product Line

The GPL was first introduced in [Lopez-Herrejon and Batory 2001] as a common computer science problem that can be tackled in terms of features so that different variants representing specialized graphs could be obtained easily. The GPL was put forth as an easy to code SPL without requiring any domain-specific expertise.

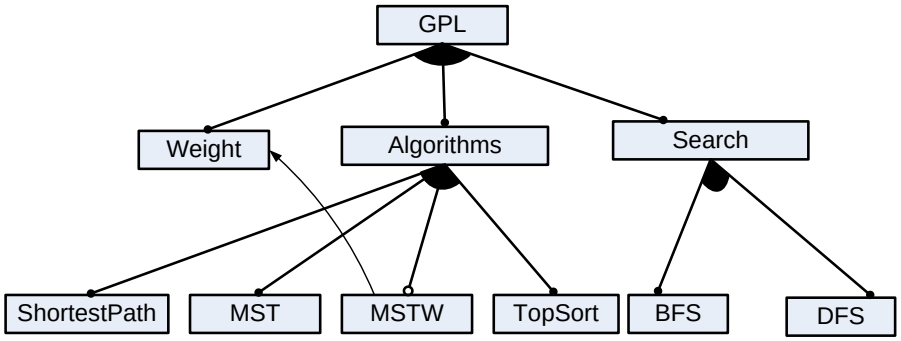

Figure 6.4: Graph Product Line in FeatureJ

Listing 6.13: Definition of productline GPL in FeatureJ

```
1  public class Launcher extends TestCase {
2   productline GPL {
3    features {
4     algorithms : more(ShortestPath, MST, MSTW?, TOPSort),
5     search  : more(BFS, DFS),
6     weight
7    }
8    more(algorithms, search, weight)
9    constraints {
10    MSTW <-> weight
11   }
12  };
13 }
```

In the FeatureJ version of the GPL, we showcase how standard coding practices such as unit testing using JUnit can be used when coding SPLs in FeatureJ. Listing 6.13 shows how the feature model shown in Figure 6.4 is represented as a productline type GPL. The GPL code base consists of 8 compilation units

namely, Graph, Vertex, Edge, Stack, Queue, PriorityQueue, Distance, and Launcher class.

Listing 6.14: Unit Testing BFS feature in a variant of GPL

```
1  public class Launcher extends TestCase {
2   variant GPL BFSGraph {
3    search = [BFS],
4   };
5   public void testBFS() {
6    GPL::BFSGraph Graph->theGraph;
7    theGraph.addVertex('A');     // 0
8    theGraph.addVertex('B');     // 1
9    theGraph.addVertex('C');     // 2
10   theGraph.addVertex('D');     // 3
11   theGraph.addVertex('E');     // 4
12   theGraph.addEdge(0, 1);      // AB
13   theGraph.addEdge(1, 2);      // BC
14   theGraph.addEdge(0, 3);      // AD
15   theGraph.addEdge(3, 4);      // DE
16   assertEquals(theGraph.bfsString().trim(),"ABDCE");
17  }
18 }
```

The JUnit TestCase class is extended by class Launcher. Listing 6.15 shows a JUnit test to check the output of breadth first search in graph that represents a variant that consists of the BFS feature from the GPL.

The line `GPL:: BFSGraph theGraph;` is transformed to the corresponding Java code that instantiates a Graph object named theGraph which is obtained using JastAdd's rewrite system to transform FeatureJ syntax to Java equivalent. Note that methods of class Graph can also be called without creating an instance first, as first shown in Listing 4.20.

Listing 6.15: Getting a Graph Object using AST Rewriting and Java Reflection

```
1  public class Launcher extends TestCase {
2   ...
3   public void testFeatureJ() {
4    // Transforming GPL::BFSGraph Graph->theGraph; to ->
5    PL GPL=new PL("GPL");
6    PLVariant BFSGraph=new PLVariant(GPL,"BFSGraph");
7    Graph theGraph= BFSGraph.getVariantObject("Graph");
8    ...
9    assertEquals(theGraph.bfsString().trim(),"ABDCE");
10  }
11 }
```

While error checking in FeatureJ ensures that variant BFSGraph will compile without errors, easy integration with JUnit, which only requires including JUnit .jar files in the classpath of the FeatureJ GPL application, enables testing that the breadth first search algorithm is implemented correctly.

More than one tests can be written that test different variants and functionality of graphs that are specific to those variants. Furthermore, a new variant can be composed by adding specific features and the functionality related to these features can be tested. By integrating error checking at the language level, it becomes possible to use error checking and testing in concert in FeatureJ as it is done with Java in which compiler checks for compilation errors and developers write tests for individual pieces of functionality.

Berkeley DB Product Line

Oracle Berkeley DB JE is a Java version of an embedded database system written entirely in Java. The Berkeley DB product line was first presented in [Kästner et al. 2007]. It is based on the concept of feature-refactoring of legacy applications [Liu et al. 2006]. Berkeley DB Java Edition was feature-refactored into an SPL using AspectJ [Kästner et al. 2007]. Current version of Berkeley DB JE is 4.0.13, while the Berkeley DB Java edition that was refactored into an SPL was 1.1.6. This version consisted of four main packages namely, je, util, compat, and bind. Collectively, these packages presented the functionality to use key/value pairs of arbitrary data based on internal B-tree implementation and provided transaction logging and cacheing facilities. The latest version of Berkeley DB Java Edition additionally provides direct persistence layers and persistent collections API.

The original feature model implicit in the refactoring of Berkeley DB JE contained 38 features. We used the #ifdef annotated version of Berkeley DB used in the original feature-refactoring as the starting point to create the FeatureJ version of Berkeley DB JE product line. We also included another package called collections available in versions 1.6.x to test a variant of this SPL against example programs given in Berkeley DB JE. This package was included as a part of the base program for completeness without feature-refactoring it. Altogether the FeatureJ version of Berkeley DB JE product line (BDBPL) consists of 57 features (including the 38 leaf features as well as intermediate features and the root feature) with 308 compilation units in 5 packages of Berkeley DB JE with 45K SLOC.

Instead of showing the feature model of BDBPL, we show the productline definition of the BDBPL in Listings 6.16 and 6.17. Listing 6.18 shows a variant of BDBPL. Listings 6.16, 6.17, and 6.18 show that productline and variant definitions of a large SPL such as BDBPL can be represented concisely and that the developer can be aware of which feature types constitute a variant even without a visual aid.

Listing 6.16: Berkeley DB productline Definition - I

```
public class BerkeleyDBLauncher  {
 productline BDBPL {
  features {
   OptionalDB? : more(Logging, Statistics?,
    DBMemoryBudgetJE?),
   Logging : more (LoggingFiner?, LoggingConfig?,
    LoggingSevere?, LoggingEvictor?, LoggingCleaner?,
    LoggingRecovery?, LoggingDBLogHandler?,
    LoggingConsoleHandler?, LoggingInfo?,
    LoggingBase, LoggingFileHandler?,
    LoggingFine?, LoggingFinest?),
   ConcurrTrans : more (DBLatchesJE?, Transactions?,
    CheckLeaks?, FSync?),
   Persistance : more (Checksum?, IIO,
    EnvironmentLocking?, IICleaner,
   Checkpointer, DiskFullError?, FileHandleCache?),
   BTree : more (InCompressor?, IEvictor?, Verifier?),
   IIO : one (SIO, NDIO),
   SIO : more (SynchronizedIO?, IO),
   NDIO: more(NIOAccess,DirectNIO?),
   NIOAccess : one(ChunkedNIO, NIO),
   Checkpointer : more (CPBytes?, CPTime?,
    CheckpointerDaemon?),
   IICleaner : more (CleanerDaemon?, Cleaner,
    DBLookAHEADCacheJE?),
   IEvictor : more (CriticalEviction?, EvictorDaemon?,
    Evictor),
   Ops : more (DeleteOperation?, RenameOperation?,
    TruncateOperation?)
  }
  more(OptionalDB, ConcurrTrans, Persistance, BTree, Ops)
 };
 constraints {...}
}
```

The original feature-refactoring of Berkeley DB JE was carried out by analyzing the domain of database engines, referring to the Berkeley DB JE manual, studying default configuration parameters and manual inspection of the source code using various Eclipse-supported static analysis facilities.

It was found in the original feature-refactoring of that there were both fine-grained and coarse-grained features. Extremely fine-grained features such as method parameters required work-arounds that would lead to the redesign of the base application. Also, optional feature problem was apparent due to interaction between directly and indirectly dependent features [Kästner 2010]. Feature interaction is defined as a situation in which two or more features

Listing 6.17: Berkeley DB productline Definition - II

```
1  public class BerkeleyDBLauncher  {
2   productline BDBPL {
3    features {...}
4    constraints {
5     Evictor <-> DBMemoryBudgetJE,
6     EvictorDaemon <-> DBMemoryBudgetJE,
7     DBLookAHEADCacheJE <-> DBMemoryBudgetJE,
8     CriticalEviction <-> InCompressor,
9     CPBytes <-> CPTime,
10    DeleteOperation <-> Evictor,
11    DeleteOperation <-> InCompressor,
12    DeleteOperation <-> DBMemoryBudgetJE,
13    DBMemoryBudgetJE <-> Evictor,
14    DBMemoryBudgetJE <-> DBLatchesJE,
15    TruncateOperation <-> DeleteOperation,
16    Verifier <-> InCompressor
17   }
18  };
19 }
```

exhibit unexpected behavior that does not occur when the features are used in isolation[Apel and Kästner 2009]. The same strategies applied to counter feature interactions in the original case-study are also applicable in FeatureJ, i.e., to separate the code of interacting features to a derivative which itself can be represented as a feature.

The treatment of nested and alternative features takes place as described earlier (cf. Section 4.2.3 on page 83). Note that while we support error checking for invalid variant types as well as any compilation errors that may occur as a result of nested and alternative feature definitions, we do not provide support for various automated refactorings as suggested in [Kästner 2010]. In FeatureJ after the errors have been found, the code needs to be redesigned manually.

Since we used version of Berkeley DB JE that was already feature-refactored, we did not face the feature-refactoring effort as documented in [Kästner et al. 2007] and [Kästner 2010]. The primary difference between the original feature-refactoring and FeatureJ version was that of the granularity of features allowed. While FeatureJ enables the same level of granularity for coarse-grained features as in virtual annotations, on the level of fine-grained features it enables containing statements rather than expressions. While it is possible to extend feature containments in FeatureJ to finer levels of granularity, we did not opt for it for the same reason as it was found in the original feature-refactoring, it would render the code base obfuscated or lead to major redesign of the code.

Since FeatureJ is IDE-independent, we believe that a sound use of FeatureJ in

Listing 6.18: A variant of productline BDBPL

```
1  public class BerkeleyDBLauncher {
2   variant BDBPL BDVariant {
3    OptionalDB = [Statistics and DBMemoryBudgetJE
4     and Logging],
5    Logging =[LoggingSevere and LoggingBase],
6    ConcurrTrans = [DBLatchesJE and Transactions
7     and FSync],
8    Persistance = [Checksum and IICleaner
9     and EnvironmentLocking and FileHandleCache],
10   IICleaner =[Cleaner and DBLookAHEADCacheJE],
11   BTree = [InCompressor and Verifier],
12   IEvictor = [CriticalEviction and Evictor],
13   Ops = [DeleteOperation and RenameOperation
14    and TruncateOperation]
15  };
16 ...
17 }
```

large legacy applications would require extending it with various existing JastAdd static AST analyses including refactoring [Schäfer et al. 2009]. Similarly, we do not provide a special consideration of feature interactions though they are clearly a major issue [Apel and Kästner 2009] as found in BDBPL [Kästner et al. 2009].

6.1.2 Conformance to Requirements of First-class Features

Our design choices in FeatureJ make it possible to express feature domain entities as domain-specific abstractions. The operations over these entities are expressed using domain-specific syntax. In the following, we elaborate on how FeatureJ specifically satisfies each requirement from the set of requirements laid out earlier in Chapter 3 towards achieving first-classness of feature domain entities:

First-classness in a Host Language FeatureJ conforms to the first requirement of representing feature domain entities with first-class status in Java as the host language. This is achieved by implementing feature domain entities as types atop the JastAdd extensible compiler system for Java. These entities are addressable at compile-time as productline, variant, and feature AST node types as described earlier in Section 4.2.1 on page 73. These are represented at run-time in terms of using objects of meta-classes PL, PLVariant, and Feature as demonstrated in Section 4.3.2 on page 90. Whereas productline type expresses the structure of an SPL and feature type enables mapping code fragments to conceptual features,

variant entities represent actual program variants and can be modified by adding or removing feature types. More than one variant types can be defined based on a productline type. This indicates that the representation of feature domain entities in FeautreJ is both addressable and manipulable as required.

Uniformity The three feature domain entities productline, variant, and feature represent feature models of the SPL under consideration and its variants, and both conceptual and concrete features respectively. The relations and constraints between conceptual features that are implicit in a feature model are made explicit in a productline type definition and utilized toward validating a variant model represented by a variant type. When more than one conceptual features map to the same concrete code fragment, this is denoted by feature containment of multiple feature entities. This representation of feature domain entities in FeatureJ satisfies the second requirement such that conceptual and concrete counterparts are represented by common language entities.

Subsumed Checking Various aspects of validation with regards to feature domain entities are part of their representation in FeatureJ. Based on common practice in JastAdd to represent extensions in terms of AST node types, the representation of various feature domain entities in FeatureJ contains corresponding validation functionality which is invoked statically during error checking to ensure productline type indicating a valid feature model, valid variant type based on a productline type that satisfies relations and constraints made explicit in the productline type. Actual program variants to be generated are checked for errors by extending error checking on per AST node basis with respect to feature containments so that accurate errors regarding feature types in individual variant types are reported.

Identity A program variant in FeatureJ is referred through a variant type at compile-time and at run-time it is executed as an instance of PLVariant class. This ensures that identity of a program variant as it was composed using feature entities is preserved. While run-time adaptation of variant types and therefore of object of the underlying application classes is not yet supported, separate namespace mechanism coupled with meta-classes based design in FeatureJ enables modifying a variant by composing another variant type as it is known which feature types make up the original variant type, and whether the modification will lead to a valid or invalid program variant based on the original feature model which is made explicit in a productline type that is available for reference as an instance of PL class.

Extensibility This requirement of first-class features is achieved by using the JastAdd extensible compiler system to implement FeatureJ. This not only enables extending Java 1.4 with FeatureJ syntax and semantics but also make possible extending FeatureJ itself to encompass Java 1.5 source with minimal effort as detailed in Section 4.3.4 on page 94. JastAdd provides a clean composition mechanism for AST, parser, and scanner

specifications that can be used to make any JastAdd-based extension of Java 1.4 and Java 1.5 feature-aware both syntactically and semantically.

6.2 rbFeatures

6.2.1 Case Studies

Table 6.2 shows each SPL, number of compilation units in each, number of features and SLOC. The number of features indicate all features including the root and intermediate features as well as the leaf features that actually contain code fragments.

The EPL presented in rbFeatures is based on an alternate feature model which is different from the one used in FeatureJ. With the EPL, we show how testing is carried out along with the application code in rbFeatures.

The Calculator Product Line (CPL) in rbFeatures showcases its use with a GUI-based SPL. We use Shoes framework for this purpose which is used in Ruby for GUI-based applications. The CPL demonstrates the code and data equivalence in Ruby and how it is utilized in rbFeatures.

We demonstrate the run-time adaptation capability of rbFeatures with the GPL. Graph application class instances exist in ProductVariant-specific scope and can be adapted to the modified structure of the ProductVariant within the context of which they exist.

The Twitter Application Product Line (TAP) case study demonstrates how rbFeatures interacts with other DSLs in Ruby to support SPL versions of web applications. Due to its language-based implementation, rbFeatures is able to make other Ruby-based DSLs feature aware so that their functionality is utilized towards achieving features of a web application (in this case, Twitter application) including communication with server, database handling, and rendering of web pages.

SPL	# Features	# SLOC
EPL	10	116
CPL	5	111
GPL	19	324
Twitter PL	9	600

Table 6.2: rbFeatures Case Studies

Expression Product Line

We use an alternate feature model of the EPL in rbFeatures from the one shown in Figure 6.5 and used as a FeatureJ case study to compare feature representation and composition. This feature model is shown in Figure 6.5. Instead of representing the functionality of the EPL in terms of printing and evaluation operations, we apply traditional domain analysis and represent data types in terms of integer numbers and addition and subtraction expressions and data operations in terms of printing and evaluation.

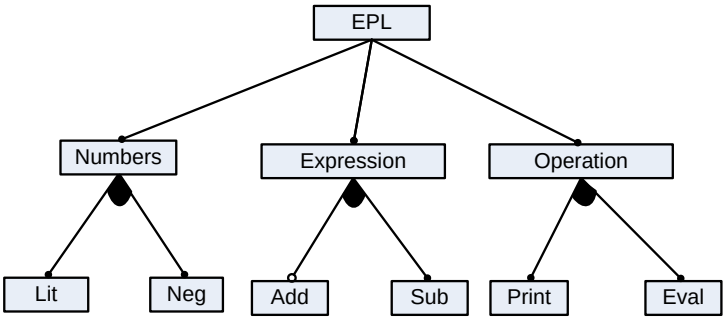

Figure 6.5: EPL in rbFeatures

The rbFeatures version of the EPL consists of 6 leaf features and 4 intermediate features including the root feature EPL. In this representation, Ruby classes Numbers, Expressions, and Operations include the Feature module. Each of the pair of classes in Lit and Neg, Add and Sub, and Print and Eval extend the Numbers, Expressions, and Operations classes. Listing 6.19 shows that the method print_eval() in class Add belongs to both the Print and Eval Feature entities. When both the Print and Eval are activated in a ProductVariant the print_eval() method is added in an instance of ProductVariant class.

Listing 6.19: Providing the Print and Eval Functionality for Add Feature

```
1 class Add
2   (Print + Eval).code do
3     def print_eval
4       print
5       Kernel.print "=#{eval}"
6     end
7   end
8   ...
9 end
```

Listing 6.20 shows how RSpec-based tests are used to code an SPL in rbFeatures as described earlier in Section 5.2.3 on page 117.

Listing 6.20: Testing Containments in `code{}` in EPL

```
it "Block containments" do
 base = lambda {
  class Print
   is Feature
  end
  class Eval
   is Feature
  end
  class Lit
   is Feature
   def initialize(val)
    raise "IntegerError" if val < 0
    @value = val
   end
   (+Print).code(self) {def print; @value.to_s; end}
   (+Eval).code(self) {def eval; @value; end}
  end
 }
 FeatureResolver.init(base)
end
```

Listings 6.21 and 6.22 indicate tests that are members of the same test suite for EPL in rbFeatures which contains test in Listing 6.20. In conjunction with the test in Listing 6.20, test in Listing 6.21 ensures that `code` blocks in Print and Eval classes (that mix-in the Feature module via Operations class) are executed with a desired output when they are activated.

Listing 6.21: Testing for Errors in EPL - I

```
it " * calling methods for activated features should have ↩
    correct output " do
 Lit.activate
 Print.activate
 Eval.activate
 Lit(1).send(:print).should eql "1"
 Lit(1).send(:eval).should eql 1
 Print.deactivate
 Eval.deactivate
end
```

On the other hand, if certain features are not activated but the methods they contain are called, then corresponding error is raised as shown in Listing 6.22.

While rbFeatures uses testing for the purposes of error checking, unlike FeatureJ this code has to be written by the developer. Although this kind of treatment of testing requires getting used to, we believe that it provides a viable option in rbFeatures toward checking SPLs for errors.

Listing 6.22: Testing for Errors in EPL - II

```
1 it " * calling methods of non-activated features should ↩
      result in FeatureNotActivatedErrors " do
2  lambda {Lit(1).print}.should raise_error(↩
       FeatureNotActivatedError)
3  lambda {Lit(1).eval}.should raise_error(↩
       FeatureNotActivatedError)
4 end
```

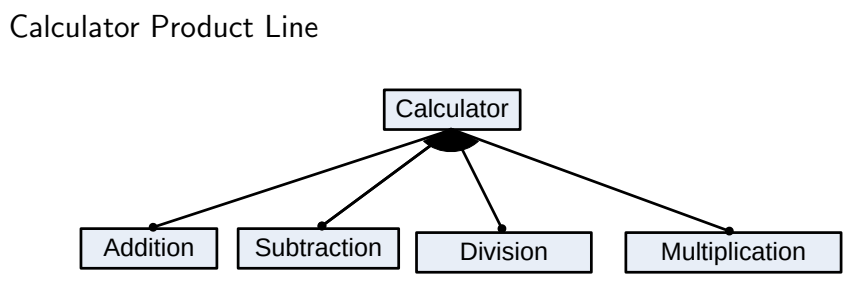

Figure 6.6: Calculator Product Line

Listing 6.23: Rendering Numbered Buttons of a Calculator

```
1 flow :width => 218, :margin => 4 do
2   \%w(7 8 9 / 4 5 6 * 1 2 3 - 0 Clr = +).each do |btn|
3     button btn, :width => 46, :height => 46 do
4       ...
5   end
6 end
7 method = case btn
8   when /[0-9]/; 'press_'+btn
9   when 'Clr'; 'press_clear'
10   ...
11 end
```

The CPL showcases application of rbFeatures to GUI applications. The CPL is based on the Shoes framework which enables creating user interfaces in Ruby [Ruby Shoes Development Community at GitHub 2009]. The feature model

Figure 6.7: Two Calculator Variants using Shoes Framework

used in CPL is simplistic. The primary difference from the GUI-based SPL in FeatureJ such as the NPL is that activation/deactivation of GUI features and corresponding functionality can be carried out dynamically.

Listing 6.23 shows how the buttons on a regular calculator are rendered in Ruby using Shoes. The name of the event to be processed is generated as a string which is called using Ruby metaprogramming methods.

Listing 6.24: Containments of CPL Features

```
ops = {}
Add.code {ops['add'] = '+'}
Sub.code {ops['sub'] = '-'}
Times.code {ops['times'] = '*'}
Div.code {ops['div'] = '/'}
ops.each do |meth, op|
  define_method "press_#{meth}" do
    if @op
      press_equals
    end
    @op = op
    @previous, @number = @number, nil
  end
end
```

Listing 6.24 shows feature containment of CPL features. Various event processing `press_{operation_method_name}` methods are generated dynamically. A controller interface is used to activate/deactivate features in CPL. Based on which features are activated, corresponding rendering and event processing code is generated dynamically and executed so that it is possible to dynam-

ically alter the appearance and functionality of a CPL variant as shown in Figure 6.7.

Graph Product Line

We have described the rbFeatures version of GPL at length throughout chapter 5 to illustrate rbFeatures syntax and semantics. Similarly, we described the origins of the GPL in Section 6.1.1 on page 132. The feature model of the GPL used in rbFeatures differs from the feature model used in the FeatureJ version of the GPL. We refer to the GPL feature model shown earlier in Figure 5.2 on page 110.

In Listing 5.17 on page 113, we showed how a ProductVariant called DFSVariant is instantiated. In Section 5.3.1 on page 118, we stated that in contrast to FeatureJ, rbFeatures enables run-time adaptation of application class objects already obtained. In Listing 6.25 we reproduce the excerpt that shows instantiation of DFSVariant.

Listing 6.25: Initializing ProductVariant DFSVariant

```
1  GPL.variant("DFSVariant").instantiate!
2  DFSVariant.class_eval do
3   graph=Graph.new
4   1.upto(5) {|n| graph + node(n)}
5   graph + edge(1 => 2, :weight => 1)
6   graph + edge(1 => 3, :weight => 2)
7   graph + edge(2 => 4, :weight => 3)
8   graph + edge(3 => 5, :weight => 4)
9   graph + edge(4 => 5, :weight => 4)
10  DFSGraph= graph
11 end
```

Note that in the DFSVariant ProductVariant, Feature shortest_path had not been activated (cf. Listing 5.16 on page 112). If we try to run the shortest_path method on the graph DFSGraph obtained in Listing 6.25, then we get the `FeatureNotActivatedError` error as shown in Listing 6.26.

Listing 6.26: Executing Non-activated Feature in DFSVariant

```
1  DFSVariant.class_eval do
2   shortest_path (DFSGraph,5)
3  end
4   # => FeatureNotActivatedError :
5   # => Feature shortest_path is not activated
```

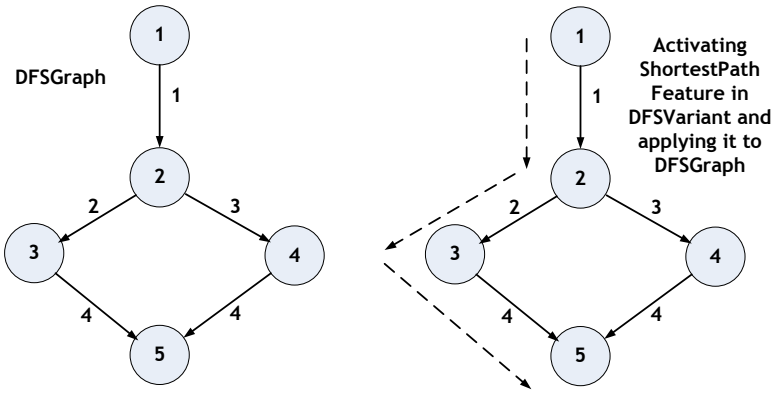

Figure 6.8: Adapting an Existing Graph Instance with DFS Feature

Listing 6.27: Adapting Existing Graph Instance to a Feature at Run-time

```
GPL.variant("DFSVariant").activate_features
 :shortest_path
DFSVariant.class_eval do
 paths=shortest_path DFSGraph, 5
 print paths
end
# => [ 1 , 2 , 3 , 5 ]
```

The run-time adaptation capability of rbFeatures is demonstrated in Figure 6.8. The Graph class instance DFSGraph indicates an application class instance that has state, including the edges between the nodes and the weights. If now we alter the configuration of DFSVariant by activating the shortest_path Feature, it is possible to apply the shortest_path method to DFSGraph because it exists within the scope of DFSVariant as shown in Listing 6.27. The left figure shows DFSGraph as it is before activating the shortest_path Feature. After activating shortest_path Feature in DFSVariant, the modifications are reflected in existing Graph class instance DFSGraph. In this regard rbFeatures differs from FeatureJ, because in FeatureJ, after modifying a variant, a Graph class instance would have to be repopulated with data for edge directions and weight, which is not required in rbFeatures because of run-time adaptation support.

Twitter Application Product Line

The Twitter Application Product Line (TAP) case study enables customizing the functionality offered by Twitter micro messaging platform in terms of number of tweets and addition/removal of registered users. The feature model of TAP is shown in Figure 6.9. The TAP showcases the application of SPL concepts to web applications. Furthermore, it is used to demonstrate how the feature domain entities represented in terms of an internal DSL in Ruby may interact with other Ruby DSLs.

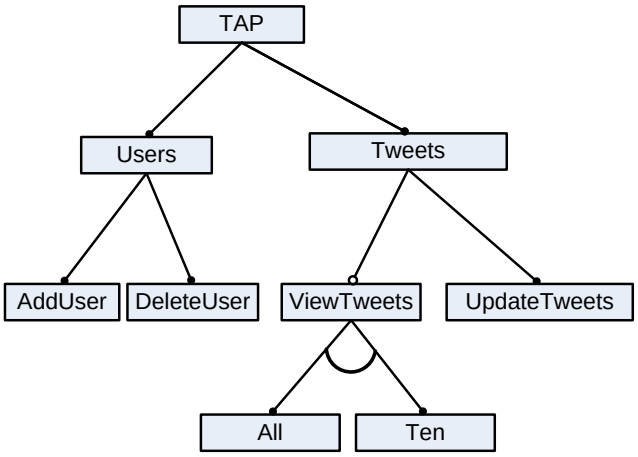

Figure 6.9: Twitter Application Product Line

The core of TAP is built around various Ruby frameworks that are made *feature-aware* so that features shown in Figure 6.9 can be selected toward a customizable Twitter messaging interaction. These Ruby frameworks are themselves represented as internal DSLs. We extend the concepts of rbFeatures as they are applied in general application to encompass these internal DSLs to make the customization of TAP variants possible. The Ruby DSLs consist of the following, which we specify along with their purpose:

- Sinatra - Sinatra is a Ruby web-framework[2]. It represents an internal DSL that can be used to configure basic properties of a web server, handle requests including query parameter parsing and responses including MIME types and dynamic templates etc.
- DataMapper - DataMapper is used to provide database abstraction[3].

[2]http://www.sinatrarb.com/
[3]http://datamapper.org/

- HAML - HAML, which stands for HTML Abstraction Markup Language, is a template engine that provides an interface to HTML with lesser number of characters than HTML[4].
- SASS - SASS, which stands for Syntactically Awesome Style-sheets, provides Ruby interface to CSS and provides the ability to dynamically update CSS properties of a web application [5].

Since all these DSLs are implemented using Ruby syntax like rbFeatures, when used in conjunction with rbFeatures, they merely represent themselves as another piece of Ruby code at the syntactic level. We only have to accommodate their semantics. We do this by integrating the semantics of these DSLs in the rbFeatures architecture which we described earlier (cf. Section 5.3.1 on page 118). Specifically the architecture of rbFeatures is extended for web applications so that following properties are supported:

- Features of a web application are defined externally in a configuration file. These features are added dynamically to ProductVariant scope that represents a variant of the web application under consideration.
- The ability to activate and deactivate features, i.e., feature configuration is supported using URLs of the form {feature_name}/{activate|deactivate}.
- Support for multiple web applications representing individual SPLs and variants of these applications is provided by the use of middleware. For this we use Rack which is a Ruby middleware and also known as *web framework framework* (the repeated used of word framework implies that Rack is framework that can be used to manipulate other web frameworks). This makes it possible to support SPLs of web applications by catering to the web application-specific requirements.

These requirements are supported by extending rbFeatures with requirement-specific functionality in Rack in what we call *RackFeature* which is a rbFeatures wrapper around Rack.

Listing 6.28 shows a scenario for adding a user. In order to add a new user to TAP, an HTTP post request to the /adduser URL is issued. This results in parsing the request for the submitted username, passing user name to the User constructer, querying the Twitter API for the username and if the user exists then adding the user to the database as shown in Listing 6.28 following which a web page is rendered (not shown in Listing 6.28). TAP itself is run with RackFeature. Thus, while RackFeature takes care of web application-specific requirements, rbFeatures is used to obtain variants.

TAP demonstrates that class-module-based representation of feature domain entities in rbFeatures enables it to blend with other DSLs which are implemented in a similar manner.

[4]http://haml-lang.com/
[5]http://sass-lang.com/

Listing 6.28: Sinatra-HAML-DataMapper-rbFeatures Interaction

```
class Tap < Sinatra::Base
  # RackFeature
  post '/adduser' do
    AddUserFeature.code do
      User.construct params['username']
      redirect '/tap/user_config'
    end
    redirect '/tap/'
  end
  get '/features' do
    @features = [AddUserFeature]
    # HAML
    haml :feature_config
  end
  configure :production do
   database = File.join(File.dirname(__FILE__),
    'lib/tap.db')
   # DataMapper
   DataMapper.setup(:default, "sqlite3:#{database}")
  end
  #  Sinatra Web Page Rendering
  ...
end
```

6.2.2 Conformance to Requirements of First-class Features

rbFeatures implements domain-specific abstractions for feature domain entities in terms of classes and modules with domain-specific syntax implemented in terms of various methods and domain-specific checking using testing. In the following, we elaborate specifically how rbFeatures satisfies the requirements of first-class features.

First-classness in a Host Language rbFeatures conforms to the first requirement by implementing feature domain entities based on the Ruby object model in which classes and objects are first-class entities. Instances of ProductLine and ProductVariant classes represent an SPL and product/variant models respectively. The features of an SPL are represented by classes which mix-in the Feature module. More than one ProductVariant instances based on ProductLine instances can exist representing multiple variants of an SPL. This class-based representation of feature domain entities coupled with metaprogramming facilities of Ruby enables addressing and manipulating feature domain entities in different situations including run-time modification of ProductVariant instances as

discussed in Section 5.3 on page 118 and demonstrated in Section 6.1.1 on page 132.

Uniformity While the instances of ProductLine and ProductVariant classes represent SPL and product/variant models, instance of classes that are mixed-in with Feature module represent both the conceptual and concrete features. This ensures that conceptual and concrete counterparts of feature domain entities remain in sync with each other during the execution of an rbFeatures program.

Subsumed Checking Due to our design choice to implement feature domain entities in terms of classes and modules in Ruby rather than extending its implementation combined with it being a dynamically typed programming language, we support checking in terms of testing as it is routinely done in Ruby. Furthermore, even though class-based representations of feature domain entities in rbFeatures interact with regular Ruby language entities, they do not affect the semantics of regular Ruby entities. While the functionality of features themselves needs to be checked with additional code, validation of a ProductVariant instance based on a ProductLine is part of rbFeatures implementation as described in Section 5.2.3 on page 117.

Identity Being implemented atop a dynamic Ruby programming language with extensive metaprogramming support that includes string manipulation and run-time evaluation, feature domain entities as they are represented in rbFeatures retain their identities throughout the execution of an rbFeatures program. By creating ProductVariant-specific execution scope and evaluating Ruby code that pertains to variant within the context of a ProductVariant class, not only run-time modification but also run-time adaptation is supported.

Extensibility The final requirement is satisfied by leveraging Ruby's malleable syntax and its extensibility mechanisms including open classes and string manipulation coupled with the ability to re-evaluate program structure. Since many other Ruby-based DSLs utilize similar mechanisms and represent extensions in term of Ruby's language entities instead of extending its C-based implementation, rbFeatures can be used comfortably alongside other DSLs.

6.3 Summary

In this chapter, we presented various SPL case studies in FeatureJ and rbFeatures and showed that individually each both FeatureJ and rbFeatures conform to all the requirements of first-class features we laid out earlier in Chapter 3. The case studies demonstrate that first-class features can be applied in to different kinds of applications including GUI- and Web-based applications and it is remarkably easy to code SPLs according to coding practices such as testing utilized in regular Java applications.

While the uniform treatment of first-class features enable run-time composition in FeatureJ, with rbFeatures run-time adaptation is possible. A cleanly separated implementation of semantics for features means that the developer need not be concerned with how FeatureJ and rbFeatures code, such as definitions of various feature domain entities, interacts with the application code. This results in concise specifications of an SPL as well as its variants.

Both FeatureJ and rbFeatures provide implementation of first-class features, but the details of implementation vary. In the next chapter we compare FeatureJ and rbFeatures to find out subtle differences that affect the implementation of first-class features in general. We then compare first-class features with other techniques we classified based on feature representation and composition earlier to show that first-class features make the process of engineering SPLs much clearer and streamlined.

Chapter 7

Discussion

> Simplicity is about subtracting the obvious and adding the meaningful.
>
> John Maeda
> *in The Laws of Simplicity*

In this chapter, first we compare first-class features as they are implemented in FeatureJ and rbFeatures. Our objective is to abstract basic ways in which first-class features can be implemented in general. We then compare first-class features with various feature modeling and implementation techniques that we classified earlier in Chapter 3 into different categories based on the nature of feature representation and composition. Finally, we present merits and limitations of first-class features.

7.1 Comparing FeatureJ and rbFeatures

While there are differences in how FeatureJ and rbFeatures conform to the requirements of first-class features, these can be boiled down to the different ways in which feature domain entities are implemented in them and the way in which host language offers certain capabilities for the same. In the following, we enlist these differences:

AST Manipulation Vs. String Manipulation

While a given programming language offers modularity mechanisms that group related code in meaning units, concrete features often cross over such modularity boundaries. Since concrete features represent disparate code fragments,

we need a mechanism given a programming language that can easily refer to such code fragments and be able to manipulate them as required. FeatureJ is based on JastAdd which treats AST as the core data structure to be used in compilation and execution of a Java program. Various code fragments in a Java program are represented in terms of AST nodes and FeatureJ interacts with the main program AST to generate program variants containing specific AST nodes representing code fragments in selected features.

Ruby on the other hand, is capable of representing a Ruby program as both an AST and as a string. While there are Ruby libraries available that represent a Ruby program in terms of s-expressions, there is native support for representing a Ruby program as a string and evaluating it. We chose string manipulation over AST representation in Ruby because of the simplicity with which parts of programs can be represented as a string, modified, and re-evaluated to adapt to the state of the running program. While Ruby metaprogramming mechanisms in Ruby also enable evaluating code fragments in a specific scope which is an ability required to address and manipulate many program variants, lack of the support for the same in Java makes the concept of namespaces necessary which we elaborate next.

Namespace Vs. Scopes

As discussed earlier, in FeatureJ it becomes necessary to implement separate namespaces for variant-specific application classes so that number of versions of the same classes can co-exist while taking care of the Java class-loading idiosyncrasies (cf. Section 4.3 on page 87). Implementing separate namespaces requires arranging variant-specific classes in different folders and loading them using custom class loaders.

In Ruby on the other hand, classes are open to re-definition either simply by re-opening them, or modifying them as strings and re-evaluating their code. Furthermore, it is possible to execute/evaluate code fragments within the context of a class using Ruby metaprogramming methods such as `class_eval`. This effectively provides means to generate ProductVariant-specific scope of application classes that mix-in the Feature module. Run-time modifications and adaptation of ProductVariant-specific objects of application classes are not visible outside this scope and thus versions of Ruby application classes that are part of other ProductVariant instances remain untouched.

Static Error Checking Vs. Testing

Java is a statically typed language whereas Ruby is a dynamically typed language. Since we express feature domain entities in terms of entities with first-class status in that language, we also follow the convention of checking that is default in the language. Implementation of feature domain entities as

AST nodes atop JastAdd compiler implementation offers opportunity to extend name/type analysis and error checking by an established pattern, i.e., feature domain entities can be checked for errors before variant programs are generated and only when the variant programs are found to contain no compilation errors can they be compiled/interpreted.

Ruby on the other hand has no compilation time and therefore the checking of feature domain entities for errors must also happen at run-time which we carry out in terms of testing.

Composition Vs. Adaptation

While Java follows a dual model of compilation to byte-code and interpretation of byte-code, Ruby code is interpreted directly. This means that the compilation and execution model of Java offers opportunities to carry out various activities involved in generating and executing program variant to spread over compile-time and run-time whereas in Ruby, everything must take place at run-time. We have seen that in FeatureJ composition of variant types is supported at both compile-time when generating program variants for the first time and run-time when modifying existing variant types. Note that in both cases, error checking that takes place is considered static because it takes place with respect to the program AST before it is transformed to byte-code, i.e., before it is compiled.

We have also seen that composition and adaptation differ in the sense that states of existing objects is not carried over when the variant to which they belong (i.e., they are objects of application classes that are variant-specific) is modified in FeatureJ. In Ruby however, it is possible to evaluate code in the context of a class or an object so that separate scope exists where modifications a in ProductVariant are immediately reflected to existing objects.

Internal Types Vs. Classes/Modules

When executing variant of a program, there is an implicit requirement to go to a meta-level. This means that using a given program, variants of that program need to be generated and execution of these program variants need to be controlled from within the original program. This implicit requirement in the treatment of feature domain entities is satisfied in FeatureJ by the dual representation of feature domain entities as internal types with their own representation in the AST and meta-classes that enable controlling their behavior at run-time. For instance, a product/variant model is represented by a variant type (that is used to generate variant-specific AST) as well as an instance of a `PLVariant` class to represent the compiled classes that are specific to this variant. In Ruby on the other hand, support for metaprogramming at the level of regular classes and modules alleviates the need for such dual representation.

Note that this point of difference is distinct from the earlier point of AST manipulation vs. string manipulation. There are two related questions: (a) whether it is possible to implement FeatureJ as an API, say only in terms of meta-classes without an internal type representation of feature domain entities and (b) whether it is possible to implement rbFeatures like FeatureJ by representing feature domain entities as Ruby types rather than in terms of classes/modules. Our implementation based on explicit requirements of first-classness and the implicit requirement of a meta-level stated above suggests that the (a) is not possible. By using only meta-classes, it is not possible to support pre-compilation checking, since instead of an AST, we would have to deal with the byte-code which is available only post-compilation. Regarding (b), while it is possible to represent feature domain entities in the Ruby interpreter implementation, such a representation would be merely redundant in case of Ruby because of its extensive metaprogramming support.

7.2 Comparing First-class Features with Other Techniques

In the following we present a comparison between first-class features and feature modeling and implementation techniques we reviewed earlier (cf. Section 3.3 on page 39). First-class features as they are implemented in FeatureJ and rbFeatures share similarities and also differ in certain respect from other modeling and implementation techniques which we explain next. Toward this end, we also recapitulate the capabilities in modeling and implementing features within the context of first-class features.

7.2.1 Feature Modeling Techniques

The capabilities in modeling features in the problem space point at the ability to choose and to represent product/variant model and constraints when representing conceptual features and proper transition between abstraction levels of features and their implementation and effectuation [Sinnema and Deelstra 2007].

First-class features provide the ability to choose by representing SPLs and features that constitute them as language entities and providing the ability to select/activate features to obtain variants. The feature domain entities are implemented at a consistent level of abstraction depending on host language and the effectuation process is built into the processing of language entities. With this, we now compare first-class features with the feature modeling techniques.

GUI-based Feature Models

Following are the similarities and differences of first-class features with feature modeling techniques that represent conceptual features as graphical entities:

Similarities The representation of variants in FeatureJ and rbFeatures is similar to decision models used in COVAMOF [Sinnema et al. 2004b], where a variant is defined within the context of an SPL. CBFM [Czarnecki et al. 2005] supports validation of product/variant models similar to FeatureJ and rbFeatures.

Differences Various feature modeling entities are graphical entities and concrete features are represented in terms of either code annotated with preprocessing directives as in pure::variants [Beuche et al. 2004], or description profiles from which a specific product/variant is generated as in COVAMOF [Sinnema et al. 2004b]. FeatureJ and rbFeatures represent all feature domain entities at the level of language.

While CBFM [Czarnecki et al. 2005] does not support error checking/testing of actual program variants, FeatureJ and rbFeatures provide support for both validation of variants and checking/testing variant programs. Finally, pure::variants, COVAMOF, and CBFM [Czarnecki et al. 2005] are all implemented as IDE-specific plugins and extensions while FeatureJ and rbFeatures are implemented atop a host language and are IDE-independent.

Features as Language Entities

First-class features share similarities in terms of language representation and differences in terms of composition with modeling techniques that represent features as language entities.

Similarities FeatureJ and rbFeatures share the basic idea of language representation of feature domain entities with techniques such as VSL [Becker 2003], AMPL [Sinnema et al. 2004b], FDL [Deursen and Klint 2002], and TVL [Boucher et al. 2010].

Differences Unlike these techniques both modeling and implementation entities are represented and at the level of language. Furthermore, composition is integrated in the representation instead of delegating it to code generation via UML as in these techniques.

First-class Architectural Variability

While first-class features share the notion of first-classness with these techniques there are certain difference as we enlist below.

Similarities The most important similarity with feature modeling languages that aim at first-class architectural variability such as VML [Loughran et al. 2008], and Koalish [Asikainen et al. 2003] is the notion of not only having a language representation of features but also with the first-class status.

Differences The most important difference is that implementation entities are components in VML and Koalish. This means that no direct treatment is provided of the capabilities of feature implementation.

7.2.2 Feature Implementation Techniques

The capabilities in modularizing and implementing features in the solution space point at being able to capture program deltas or code fragments from a wide range of granularity, identifying and possibly naming them when representing concrete features and being able to compose them flexibly, without dependence on order, and with support for checking the variants to be generated [Lopez-Herrejon et al. 2005a].

First-class features as they are implemented in FeatureJ and rbFeatures support wide range of granularity of code fragments (in FeatureJ, granularity supported is statement-to-entire classes and in rbFeatures any code from expression to entire programs can be contained in features), name and identify all feature domain entities, do not depend on ordering when generating variants, can be composed at both compile-time and run-time (and in case of rbFeatures can be adapted at run-time) and finally, provide support for checking (validity of variants as well as compile-time errors in variants to be generated in FeatureJ and validity of variants and testing of SPL code in rbFeatures). With this we now compare first-class features with feature implementation techniques.

Features in terms of Preprocessors and Annotations

First-class features share similarity with these techniques in terms of how code fragments are contained in concrete features and differ in most other respects as enlisted below.

Similarities The feature containments of concrete features in FeatureJ and rbFeatures are similar to #ifdef preprocessor directives [Spencer and Collyer 1992] and virtual annotations [Kästner and Apel 2009] in terms of containing code fragments.

Differences As far as feature containments in FeatureJ and rbFeatures are considered, they are part of the syntax and semantics of the host language rather than being anchors for including or excluding textual code similar to #ifdef preprocessors [Spencer and Collyer 1992] or being virtual in nature [Kästner and Apel 2009].

FeatureJ and rbFeatures differ further from these techniques in terms of the nature of the implementation of feature domain entities, as internal types and meta-classes and classes/modules and metaprogramming instead of language-based [Spencer and Collyer 1992] or IDE-based [Kästner and Apel 2009] external processing of only the concrete features.

First-class features also differ from virtual annotations in terms of representation of SPL and product/variant models as language entities instead of being IDE-based views.
While automated refactoring of code in situations such as multiple/nested containments and alternative features is supported by virtual annotations [Kästner 2010], first-class features do not support them. Finally, virtual annotation support checking which is solidified with a formal calculus called Colored Featherweight Java (CFJ) and type rules [Kästner and Apel 2008], whereas first-class features extend name/type analysis and error checking of host language compiler in case of FeatureJ and utilize testing as an indirect means to support checking in case of rbFeatures.

Features as Units of Modularity in a Given Modularity Mechanism

While concrete features are implemented in both these techniques and first-class features, the differences outweigh the similarities.

Similarities The similarity between techniques which implement concrete features such as in AspectJ via aspects [Kästner et al. 2007], in ObjectTeams with teams and roles [Hundt et al. 2007; Lopez-Herrejon et al. 2005a], in Jiazzi via atoms and units, in Scala via traits, in HyperJ via hyper-slices [Lopez-Herrejon et al. 2005a] and in CaesarJ with aspects and roles [Mezini and Ostermann 2004] etc., and FeatureJ and rbFeatures is that concrete feature are implemented with special consideration.

Differences Two important differences between these techniques and first-class features are that, first, all feature domain entities are represented instead of only the concrete features and second, unlike an indirect representation of features employed by these techniques, concrete features as well as the rest of the feature domain entities are first-class in their status.

Features as Explicit Language Entities

While first-class features share the idea that features should be made explicit as in these techniques, the differences arise due to the first-class status of all feature domain entities.

Similarities First-class features are similar to these techniques in that features are explicit language entities.

Differences Compared to the feature implementation techniques that only represent concrete features (in terms of a dedicated feature modularity mechanism of refinements) [Apel et al. 2009b; Batory 2004], FeatureJ and rbFeatures represent all feature domain entities as language constructs

including the SPL and product/variant models which are represented via folder hierarchies in these techniques.
The order of refinements in refinement chain is important in these techniques and requires to be carefully stated in the equation files to avoid compilation errors.
FeatureC++ uses references to the feature model of an SPL stored in XML format to enable composing variants dynamically and provides validation checks for such dynamically composed variants. While the SPL and product/variant models are expressed in terms of an API, they are not integrated into the language. When using a reference to the external feature model, the refinements still have to be stated in a specific order by the developer. The variants to be generated are not checked for compile-time errors. Furthermore, use of decorators for dynamic composition forces the developer to take care of inheritance ordering when dynamically composing multiple variants [Rosenmüller et al. 2008b; 2009] while in FeatureJ and rbFeatures semantics of features is localized in the representation of feature domain entities and their composition which requires no special consideration on the part of the developer.

7.2.3 Traceability in Feature Modeling and Implementation Techniques

We compare traceability as it is supported by feature modeling and implementation techniques and by first-class features separately, because of the special consideration given to it in the design of first-class features by presenting feature domain entities uniformly. We established various traceability paths as the desirable properties (cf. Section 3.2.3 on page 34). In our analysis we found that in most feature modeling and implementation techniques, traceability is retrofitted rather than considered beforehand. We have seen that among feature modeling and implementation techniques some of the traceability properties are supported in pure::variants with IBM Rational DOORS® connector [Systems 2010], in COVAMOF with Variability View [Sinnema et al. 2004a], in Koalish with the Kumbang configurator [Asikainen et al. 2003], in virtual annotations using views [Kästner 2010], and in FeatureC++ using XML-based description of an SPL at run-time [Rosenmüller and Siegmund 2010; Rosenmüller et al. 2008a].

By the uniform representation of various feature domain entities in the implementations of first-class features, we support all traceability properties. Traceability between the conceptual and concrete features, between SPL and product/variant models and between SPL and product code and implementation artifacts are made available by feature domain entities integrated into a host language as first-class entities with SPL and variant programs linked to SPL and variant entities. Our design of feature domain entities makes sure that each of feature, SPL, and variant entity is interconnected and can always refer

to its implementation counterparts as well as its relation to other entities. We also use this ability when composing variants at run-time (in FeatureJ) and when adapting application class objects to variant modifications at run-time (in rbFeatures).

7.3 Merits

In this section, we first enlist merits of first-class features as an FOSD technique followed by explanation of how advantages of separation of concerns are achievable by first-class features.

7.3.1 Merits as an FOSD Technique

In the following we enlist what we think to be the most important benefits of first-class features as demonstrated in our implementations and case-studies. We attribute these merits to the special treatment of representation and composition of feature domain entities in first-class features.

Clean Separation of Feature Concern

One of the primary advantages of first-class features is the clean separation of features as a kind of concern. This is achieved by giving features their own syntax and self-contained semantics. First-class features are implemented with minimally invasive semantics, i.e., they do not alter the semantics of existing language entities in the host language like other techniques (as exemplified by the implementation of EPL in FeatureJ in Section 6.1.1 on page 122). Similarly, in spite of the very nature of concrete features, i.e. syntactically they are spread all over the code, the syntax for feature containments does not leave the code unreadable.

Coherent Treatment of Feature Domain Entities

The combined and uniform treatment of concepts in the problem and solution spaces means that the conceptual integrity is preserved all the time, from analysis phase to testing phase of SPL development. All the crucial components of the domain of features, i.e., all feature domain entities, are uniformly accessible and manipulable. This means that it can be easily comprehended what is going on at any stage of SPL development. Some feature modeling techniques must rely on correct working of external elements such as UML code generation and some feature implementation techniques must depend on the SPL developer's skill to ensure that concrete feature are consistent with the feature model of the SPL under consideration. The coherent treatment of feature domain entities in

first-class features helps to cleanly distinguish between the responsibilities of the developers and that of the technique. The SPL developer defines an SPL and product variants and indicates which code fragments constitute which features. The technique automates variant generation/modification. The SPL developer need not worry about the semantics of the host language or correctness of SPL and product/variant models or generated variants because first-class features take care of these responsibilities.

Support for Error Checking and Testing

In the analysis of capabilities of features from modeling and from implementation perspective, we found that only a handful of techniques provide support domain-specific checking of feature domain entities [Kästner 2010; Thaker et al. 2007]. Furthermore, often the decision to include checking is taken a-posteriori. This means that domain-specific checking is not considered from the beginning and as a result it becomes quite difficult to offer checking support later.

First-class features have not been formalized yet. But in FeatureJ we rely on the extensive error checking in JastAdd which is highly compliant with JLS [Ekman and Hedin 2007c] and which is extended on per node basis to check errors in variants that are defined in the program. The fact that base compiler (Java 1.4) checking is extended in FeatureJ helps us in easily extending FeatureJ's ability to check Java 1.5 programs with similar capability for reporting variant- and productline-specific error reporting. Similarly in rbFeatures we use testing for validating and checking variants which adjusts nicely with other Ruby-based DSLs which also use testing for the same purpose.

Clear Traceability Links

Like domain-specific checking, traceability is also usually thought of later rather than including it in the design of feature modeling/implementation technique. In first-class features, conceptual and concrete entities are one and the same. This means that traceability is an inherent attribute of first-class features. Furthermore, support for identity means that even post-composition, it is known which features were selected in a variant, which code fragments each of these features consisted of and to which SPL these features belonged to.

Feature-awareness and easy interoperability of DSLs

The fact that first-class features are integrated in a host language in an extensible manner means that other extensions created using the same extensibility mechanism can be made feature-aware with the minimum of efforts. This was exemplified in FeatureJ by making Java 1.5 language extensions aware of

features. While being considerable in terms of code size, this process was conceptually straightforward. The complexity of this effort was also reduced by the fact that semantics of Java language entities was never altered which would otherwise be a major concern.

Similar to extensions, it is also possible to interoperate with other DSLs that are based on the same host language as demonstrated in the TAP case study in rbFeatures. This is made possible again by language integration of feature domain entities and their self-contained semantics. The fact that same patterns of extensibility are followed by FeatureJ and rbFeatures as other extensions/DSLs in Java and Ruby respectively, makes interaction between other extensions/DSLs and implementations of first-class features uncomplicated and easily achievable.

7.3.2 Achieving the Merits of Separation of Concerns with First-class Features

Recall that we began this dissertation with the discussion of the essential difficulties involved in creating good software and how the principle of separation of concerns enables tackling these difficulties (cf. Section 2.1 on pages 8–10). While we have discussed the specific merits of first-class features as an FOSD technique in the preceding section, in this section our objective is to relate the solutions of separation of concerns to first-class features toward resolving the essential difficulties of creating good software in general and good feature-oriented software in particular. In our opinion, first-class features enable achieving the merits of separation of concerns in the following manner:

Customization to address conformity in SPLs We have demonstrated the ability of first-class features toward flexible customization in SPLs in terms of composition and adaptation possibilities for an SPL and its variants. Using first-class features, it is possible to declare and modify variants at different times during the execution. Furthermore, different variants may be defined and used that cater to specific stakeholders and their requirements.

Maintenance to address complexity in SPLs Integration into a host language with first-class status as well as contained semantics offered by leveraging host language's extensibility mechanisms means that first-class features can enable disciplined maintenance to tackle complexity of creating and maintaining SPLs. Although we have not studied the effects of evolution of an SPL on first-class features or vice-versa, we believe that because first-class features do not affect the semantics of regular host language entities, it is possible to evolve the underlying application and the SPL based on it separately and without introducing complications in either.

Reuse to address changeability in SPLs Because first-class features are entities of their own right in a host language, we believe that they enable reuse in a variety of ways toward addressing changeability in SPLs. For instance, in an extreme case, it is possible to specify different SPLs with their features and their variants in the same application, where same code fragments contribute to multiple features from different SPLs. While we have tested such capability on smalls applications such as e.g., the notepad application, we acknowledge that further experiments may be required to test how first-class features fair in case of reasonably large applications with multiple SPLs and their variants.

Comprehension to address invisibility in SPLs Particularly two facets of first-class features enable better comprehension, namely self-contained semantics and implicit traceability links. We believe that when using first-class features, an SPL developer can easily maintain the link between his intention in conceptualizing a characteristic of a software system as a feature, implementing it, and reusing it, all the while keeping his intentions and the code comprehensible to other developers.

We have seen that first-class features offer many interesting possibilities, yet there is always room for improvement as we describe in the next section.

7.4 Limitations

In the following we discuss what we think to be the limitations of first-class features.

Support for Non-code Artifacts

Until now, in our implementations of first-class features as well as case studies, we have not considered composition of non-code artifacts. Examples of non-code artifacts include build files and documentations or docs among others. The techniques that do consider non-code artifacts use folder hierarchies for this purpose which are also used to arrange features [Batory 2004]. One general way to approach the treatment of non-code artifacts would be to give a programmatic access to them and include the statements of such interface in the definition of features. Essentially, the access to non-code artifacts would be codified and used. We discuss how this could be done for Ant build files and documentation APIs in the following:

Ant build files The access to Ant build files can be provided through the Apache Ant API [Moodie 2006][Chapter 12]. Such a programmatic build would assign specific task in a programmatic project to specific features. In this case programmatic means that a project and a task are represented

at the language level through their counterparts in the related API. For example, a project is represented by org.apache.tools.ant.Project class and deploying and un-deploying a task is carried out using DeployTask and UndeployTask classes in the org.apache.catalina.ant package. Being part of definition of features, specific tasks in a project would be selected to be deployed in relation to a variant. In Ruby, Rant is a Ruby API that can be used to control builds since build-specifications are pure Ruby code.

Documentation Like build files, document generation from doc comments/-tags in the code can be controlled programmatically through a doc API. In Java, examples of these kind of API include JavaDoc and XDoclet [Hightower et al. 2004] while in Ruby, RDoc provides API for document generation. The statements related to documentation to be generated for specific code entities can be included in specific features. Thus a variant would consist of documentation for code fragments that were selected in it.

Note that there are APIs in Java and Ruby for major non-code artifacts. One problem we perceive in using programmatic access to include non-code artifacts is when feature-refactoring legacy applications which may already posses legacy non-code artifacts. We believe that the effort to arrange non-code artifacts in a legacy application in a folder hierarchy (such as when using AHEAD which specifically considers the treatment of non-code artifacts [Batory 2004]) and effort to give a programmatic access and include it in variant generation are at par. In future, we would like to investigate in these directions.

Formalization of First-class Features

While we have presented multiple implementations of first-class features in disparate programming languages along with case studies, we have not yet formalized first-class features. We showed how name/type analysis and error checking of the host language compiler itself can be extended to accommodate feature domain entities in FeatureJ in which related attributes in JastAdd's Java 1.4 and 1.5 compilers are extended. In rbFeatures, while product/variant models could be checked for validation with respect to the feature model of the SPL on which it is based while rest of the aspect are checked via testing. In both the cases, the stress is on checking only the variants that are defined in the program rather than checking all variants.

Other techniques that support error checking an SPL check all the variants at one go using the entire code base of an SPL rather than individually checking each valid variant. The propositional calculus based checking is justified based on a formal model of the way features are represented in a given technique, e.g., features as refinements [Thaker et al. 2007] and the related type system [Delaware et al. 2009] and virtual annotations and the related type system

[Kästner 2010; Kästner and Apel 2008]. In general, these formalizations are based on subset of Java such as Lightweight Java (LJ) [Strniša et al. 2007] and Featherweight Java (FJ) [Igarashi et al. 2001] which are reduced languages that drop most features of a full language to enable rigorous proofs for type safety with only core features of Java.

We believe that when considering the formalization of first-class features, following challenges will have to be addressed:

Representing all feature domain entities We will need to represent all feature domain entities in a reduced language such as LJ or FJ. A related approach is that of FFJ_{PL} which extends Feature Featherweight Java (FFJ) which is a reduced language targeting a type system for refinement-based features [Apel et al. 2008a] and extended to include checking of SPLs [Apel et al. 2009a; 2010]. The design decision in FFJ_{PL} is to use feature models only to check that features and specific program elements are present in various circumstance and this is carried out by using functions and predicates [Apel et al. 2009a]. We however, would like to investigate actual representation of all feature domain entities.

Reconciling static and dynamic language properties While the requirements of first-class features are language independent such that they can be implemented in any language as shown by Java-based FeatureJ and Ruby-based rbFeatures, this presents a difficult scenario of whether it is possible to formalize both static and dynamic implementations.

In future, we would like investigate in these directions to present a formalization of first-class features.

7.5 Summary

This chapter presented a comparison of FeatureJ and rbFeatures in order to obtain basic ways in which first-class features can be implemented. We showed that the ability to manipulate a program either as an AST or a string is necessary along with mechanism to separate namespaces and scopes. We also discussed domain-specific checking and composition and adaptation in both FeatureJ and rbFeatures as well as the representation of domain-specific abstractions in them.

We then compared first-class features to other FOSD techniques and discussed the similarities and differences between them. First-class features offer many merits as an FOSD technique which we discussed next along with their ability to achieve advantages of separation of concerns in creating feature-oriented software. Finally we discussed what we think to be limitations of first-class features and indicated important considerations in supporting related capabilities in future.

Chapter 8

Concluding Remarks

> Most creativity is a transition from one context into another where things are more surprising. Theres an element of surprise, and especially in science, there is often laughter that goes along with the "Aha". Art also has this element. Our job is to remind us that there are more contexts than the one that we're in the one that we think is reality.
>
> Alan Kay
> *in A Conversation with Alan Kay - ACM Queue*

In this chapter, we summarize the dissertation and its contributions. We lists suggestions for further work related to first-class features and their implementations. Finally, we provide perspectives for FOSD in general based on our experience with first-class features and their implementations.

8.1 Summary of the Dissertation

We discussed in Chapter 3 the dual nature of features as conceptual and concrete entities in problem and solution spaces respectively. We reviewed the current state of the art in capabilities of features from modeling and implementation perspectives. We showed that traceability properties are not considered in feature modeling and implementation techniques and proposed another view on the mapping between problem and solution spaces called traceability view apart from the dominant configuration and transformational views. We derived a problem statement with regards to challenges in FOSD toward unifying feature modeling and feature implementation. The research questions that we coined, target the representation and composition mechanisms related to specific feature domain entities. As a step toward the solution, we classified

various feature modeling and implementation techniques into three categories each based on the representation of feature domain entities. With the help of modeling, implementation, and traceability properties, we analyzed each category of technique in terms of which feature domain entities are represented and consequences for the composition mechanism. We used this analysis to propose a new representation of features called *first-class features* in terms of a set of requirements and showed how these requirements can be implemented in a given host language.

In Chapter 4, we elaborated how the requirements of first-class features are implemented in FeatureJ. We first discussed JastAdd, the extensible compiler system for Java, in detail. JastAdd enables composition of new Java language extensions in terms of parser, scanner, and AST specifications. We described oft-used patterns of extensibility in JastAdd which we used in the implementation of FeatureJ. We described the productline, variant, and feature types in FeatureJ in terms of extended Java syntax and semantics. The name/type analysis and error checking in JastAdd Java compilers is extended to support compile-time error checking of individual variant types and variant programs to be generated based on them. Finally, we explained the architecture that enables us to put all parts of FeatureJ together and showed that FeatureJ implementation for Java 1.4 can be extended in a systematic manner to encompass Java 1.5 language entities.

In Chapter 5, we elaborated how the requirements of first-class features are implemented in rbFeatures. We first discussed the object model of Ruby and language entities such as modules, callable objects, and open classes that are important from point of view of supporting implementation of feature domain entities. We then explained Ruby's metaprogramming and string manipulation capabilities which are used extensively for variant generation and modification in rbFeatures. We then explained ProductLine and ProductVariant classes and Feature module that represent feature domain entities in rbFeatures. Finally, we discussed the architecture of rbFeatures that enables run-time composition of ProductVariant instances and adaptation of ProductVariant-specific application class objects.

We presented in Chapter 6 four SPL case studies each in FeatureJ and rbFeatures ranging from small to large code sizes. We modeled the EPL in FeatureJ as specified originally in [Lopez-Herrejon et al. 2005b]. We compared representation and composition of feature domain entities in other feature implementation techniques as exemplified in [Lopez-Herrejon et al. 2005b] with the implementation in FeatureJ and showed that FeatureJ version of EPL is semantically less invasive, easy to express and comprehend, capable of handling various feature granularity levels, and supports extensive manipulation of various feature domain entities. We also discussed other case studies in FeatureJ such as NPL, GPL, and BDBPL to demonstrate application of FeatureJ to GUI applications, easy integration of testing along with error checking, and concise representation of SPLs and variants of large applications. We then discussed

an alternate feature model for the EPL when implementing it with rbFeatures as well as application of rbFeatures to GUI applications in Ruby with the CPL. We showed in the GPL implementation of rbFeatures that it is possible to alter and adapt the configuration of variants at run-time and apply modified features to existing application class objects within the context of a variant. The TAP demonstrated that rbFeatures can be used in a seamless manner along with other Ruby DSLs to create SPLs of web applications. Finally, we explicitly discussed the conformance of requirements of first-class features in FeatureJ and rbFeatures.

In Chapter 7, we compared FeatureJ and rbFeatures and found that supporting variety of granularity levels in non-standard grouping of code fragments as well as variant generation/modification mechanism requires stepping outside the regular compilation and interpretation models. This is achieved in FeatureJ by AST manipulation with namespaces and rbFeatures by treating and manipulating program elements as a string and using metaprogramming mechanism to create specialized scopes. Furthermore, we found that compilation and interpretation models of Java and Ruby affect both the way in which feature domain entities are represented as well as the way in which they are composed. We then compared first-class features with the categories of feature modeling and implementation techniques put forth earlier in Chapter 3. We showed that while first-class features as implemented in FeatureJ and rbFeatures share some similarities with these techniques, they differ from most techniques in terms of uniform and first-class representation of feature domain entities and their composition as an extension of compilation/interpretation in the host language. First-class features prove advantageous in terms of cleanly separated semantics of feature concern with clear traceability links and enable making other extensions and other DSLs feature-aware in a straightforward way. Finally, we discussed the limitations of first-class features in terms of lack of formalization and explicit support for non-code artifacts.

8.2 Contributions

This dissertation contributes to the research on features and FOSD in general on two levels, namely conceptual and concrete[1]. First, we reviewed the concept of features, traced and explained its dual nature, examined its modeling, implementation, and traceability properties, and proposed a new representation called *first-class features*. We elaborated how this representation can be implemented in terms of a set of requirements. These requirements assure that first-class features refrain from viewing features from one side only and offer the best of both modeling and implementation worlds. This is achieved all the while enabling traceability and leaving room for extensibility.

[1] This pun is deliberate.

While the first contribution is conceptual in nature, the second is concrete. FeatureJ which is an implementation of first-class features in Java demonstrates that first-class features can be implemented across an entire static language (Java 1.4) including its extension (Java 1.5) as described at length in Chapter 4. We also elaborated rbFeatures in Chapter 5 and contrasted it with FeatureJ in Chapter 7, and demonstrated that first-class features can be easily implemented in dynamic languages as well. Our work reveals interesting counterpoints and guideposts for implementation of first-class features in static and dynamic programming languages.

Our case studies confirm the utility of first-class features in terms of unified representation and coherent composition that facilitates maintenance. SPLs developed using first-class features can be customized in a straightforward way without being concerned with the semantics of the host language. First-class features as they are implemented are easy to comprehend and reuse because of the explicit connection between the conceptual and concrete feature domain entities with clear traceability links.

8.3 Future Work

We have already indicated a few directions along which future work related to first-class features could be carried out, namely explicit support for non-code artifacts and formalization of the concept. At the same time, we are involved in other works some of which are in progress while plans for the other are on the table. In the following, we take review of such future work:

Run-time Adaption in FeatureJ We discussed earlier that while FeatureJ supports run-time composition of variants, it does not support run-time adaptation (cf. Section 4.3.2 on page 90). We have proposed to integrate an existing run-time adaptation approach in Java with FeatureJ [Sunkle and Pukall 2010]. The run-time approach for Java is based on an Eclipse plugin that uses JVM tool interface to obtain information about alive objects and their classes in a running application [Pukall et al. 2008; 2009]. It is based on first implementing class schema changes by creating new classes with changed schema and renaming them. Schema changes means that for example, one or more methods of original and updated class differ. Then, it changes all calls to the instances of classes that have been updated, creates an instance of each updated class, and finally, maps the states of old callee instance to the newly generated instances. Note that it is precisely the last step that is missing in FeatureJ. The complex details of state mapping are taken care of by the Eclipse plugin for which the input is original and modified versions of classes which can be easily provided via PLVariant instances in FeatureJ. In future, we wish to integrate this approach to enable adaptation of application class objects when configuration of a variant is changed at run-time by adding

or removing features and then test adaptation capability with the existing case-studies.

Feature Modeling Extensions in FeatureJ and rbFeatures While in both FeatureJ and rbFeatures we represent feature models as suggested in [Kang et al. 1990], there is a possibility to include further modeling extensions in the syntax and semantics of both FeatureJ and rbFeatures. Following the basic structure of a feature model suggested in [Kang et al. 1990], a variety of feature modeling extensions have been proposed such as *feature groups* where non-leaf features are treated as grouping features [Riebisch et al. 2002], *feature cardinalities* using which one or more instances of a feature can be indicates such that optional and mandatory features become special cases [0..1] and [1..1] and *group cardinalities* such that features to be selected from a group can be indicated in terms of any given cardinality such that the original inclusive, inclusive-or, and exclusive-or relations (indicated in FeatureJ and rbFeatures by various operators) become special cases of group cardinalities [Czarnecki and Kim 2005; Czarnecki et al. 2002; 2005; Kim and Czarnecki 2005], *feature attributes* that represent choice of value from infinite domain such as integers [Czarnecki et al. 2002], and *feature diagram references* which enable referring to other feature diagrams/models [Czarnecki et al. 2004] including recursive references.

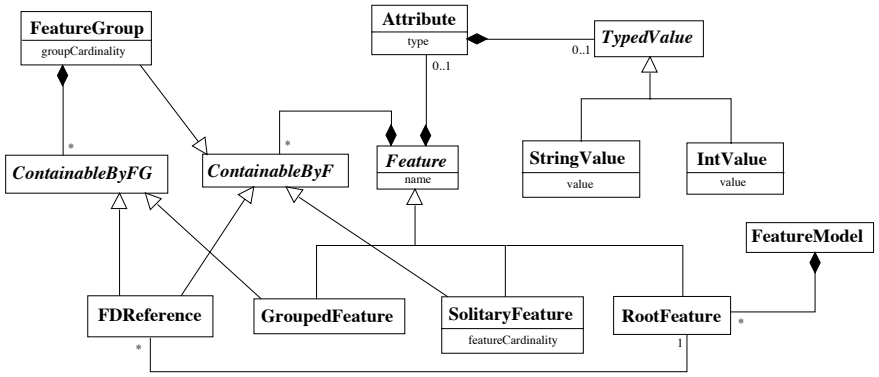

Figure 8.1: UML Metamodel for Feature Modeling Extensions [Czarnecki et al. 2004]

To explain these extensions, a UML metamodel is suggested in [Czarnecki et al. 2004] shown in Figure 8.1. The most important fact in this metamodel is that `Feature` is an abstract class to which all other entities are related by inheritance and association. We refer to the type hierarchy of feature domain entities in FeatureJ (cf. Figure on page 74) and the relationship between feature domain entities in rbFeatures (cf. Figure 5.7

on page 117). We designed the structure of feature domain entities in our implementations without being aware of a metamodel such as the one shown in Figure 8.1, yet there is a striking similarity between this metamodel and the way we structured various feature domain entities. Our design of FeatureJ and rbFeatures was driven by the requirements of first-class features and extensibility mechanisms available in JastAdd and Ruby respectively.

The feature, SPL and variant entities in our implementations correspond to the `Feature`, `FeatureModel` (as well as a `FDReference`), and instances of `FeatureModel` based on cardinality where `Feature` is the central entity in Figure 8.1 on the preceding page. This correspondence paves way for us to easily represent groups, cardinalities, and attributes as extended AST nodes in FeatureJ and as classes and modules in rbFeatures by extending our implementation in a straightforward manner. While the usability of individual feature modeling extension has been corroborated when modeling[Czarnecki et al. 2004], we will be able to gauge their impact at syntactic and semantic levels in FeatureJ and rbFeatures as a part of future work.

8.4 Perspectives

The author of the dissertation considers himself very fortunate to have had the opportunity to carry out research in a relatively new and extremely vibrant field of study that is features and FOSD. During our tryst with a multitude of ideas, we often came across deeper issues which indicate how essential it is to go back to the basics. We wish to discuss the same at the conclusion of this dissertation.

8.4.1 Verification in FOSD

> My original postulate, which I have been pursuing as a scientist all my life, is that one uses the criteria of correctness as a means of converging on a decent programming language design - one which doesn't set traps for its users, and ones in which the different components of the program correspond clearly to different components of its specification, so you can reason compositionally about it.
>
> Charles A. R. Hoare
> *in Oral history interview with Charles Antony Richard Hoare*

The idea of verification which is defined as the *confirmation, through the provision of objective evidence, that specified requirements have been fulfilled*

[ISO/IEC 2005] or *a formal proof of program correctness* [ISO/IEC 2009], has been around for decades [Hoare 1972]. Verification has been recently seeing light of the day in software engineering in general [Hoare and Misra 2005] and FOSD in particular [Batory and Börger 2008]. In our requirements for first-class features and our implementations we observed that integrating certain characteristics into the design of a technique for FOSD enabled us to avoid perils of making workarounds afterwards. We believe in the idea that composition and verification should proceed hand in hand [Xie and Browne 2003] and representation (of concerns) could be made to aid verification [Leavens et al. 2006]. There are already some results related to verification in FOSD [Börger and Batory 2008] and we think that this will be a major direction of research in FOSD in coming years.

8.4.2 Language Vs. Design in FOSD

The issue of whether it is the programming languages or it is the design that addresses problems in software engineering has been longstanding. The following two quotes by Niklaus Wirth, who has been instrumental to the field of software engineering in his role as chief designer of programming languages such as Euler, Algol W, Pascal, Modula, Modula-2, Oberon, Oberon-2 [Wirth 1985], and David Parnas, who gave the principle of separation of concerns its due and fathered the concept of program families [Parnas 1972b; 1976] indicate the two sides of this issue.

> We know that it is better if each basic concept is represented by a single, designated language construct.
>
> Niklaus Wirth
> *in A few words with Niklaus Wirth*

> Wirth is best known for his work as a designer of languages, so it is not surprising that he views the problems of software design as a question of language.
>
> David Parnas
> *in Why software jewels are rare*

We restrict the scope of this dilemma to FOSD. We have seen that while the concepts of FOSD can be traced back to program families and separation of concerns, ultimately the concerns must be represented in a programming language so that a software could be decomposed along the dimensions of concern representation and later composed again [Tarr et al. 1999]. The fact that no matter whatever the design principle, it must be expressed in a concrete language form indicates that it is the programming languages where the effort must be made to enforce any design principle.

While it is possible that the author's background in the composition of language features might have been an influence [Sunkle 2007; Sunkle et al. 2008a]

with regards to the way features are treated, the connection between separation of concerns and the degree to which a programming language supports specific concerns has been acknowledged by nearly every researcher interested in separating concerns and they do so at the programming language level [Aksit 1996; Aldrich 2000; Ernst 2003; Hürsch and Lopes 1995; Kiczales and Mezini 2005; Lai et al. 2000; Nierstrasz and Achermann 2000].

In retrospect, this dissertation began with the principle of separation of concerns which is a *design* principle and our end products are FeatureJ and rbFeatures, which are domain-specific *languages* designed and implemented based on our requirements of first-class features. On the other hand, it seems that treating features from both a design perspective and from a programming language perspective (including IDE/Tools related techniques) is required and advantageous [Batory 2009]. The final verdict on this issue is far from out which makes this field of study all the more interesting.

Bibliography

J. Åkesson, T. Ekman, and G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75 (1-2):21 – 38, 2010. ISSN 0167-6423. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

M. Aksit. Separation and Composition of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4es):148, Dec. 1996.

J. Aldrich. Challenge Problems for Separation of Concerns. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, Oct. 2000. ACM Press.

V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace. Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems. In D. Benavides, A. Metzger, and U. W. Eisenecker, editors, *VaMoS*, volume 29 of *ICB Research Report*, pages 9–17. Universität Duisburg-Essen, 2009.

S. Apel. *The Role of Features and Aspects in Software Development.* PhD thesis, School of Computer Science, University of Magdeburg, Mar. 2007.

S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 59–68. ACM Press, 2006.

S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Guest Column.

S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 122–131. ACM, 2006. ISBN 1-59593-375-1.

S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 101–112. ACM, 2008a. ISBN 978-1-60558-267-2.

S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In J. Meseguer and G. Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008b. ISBN 978-3-540-79979-5.

S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-Safe Feature-Oriented Product Lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009a.

S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-independent, Automated Software Composition. In *ICSE*, pages 221–231. IEEE, 2009b. ISBN 978-1-4244-3452-7.

S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-Safe Feature-Oriented Product Lines. *Automated Software Engineering – An International Journal*, abs/1001.3604:1–50, February 2010.

M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. ContextJ - Context-oriented Programming for Java. *Computer Software of The Japan Society for Software Science and Technology*, June 2010.

M. A. Ardis and D. M. Weiss. Defining Families: The Commonality Analysis (Tutorial). In *ICSE*, pages 649–650, 1997.

T. Asikainen, T. Soininen, and T. Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In F. van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer, 2003. ISBN 3-540-21941-2.

P. Avgustinov, T. Ekman, and J. Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In T. D'Hondt, editor, *AOSD*, pages 25–35. ACM, 2008. ISBN 978-1-60558-044-9.

J. Bachrach and K. Playford. The Java Syntactic Extender. In *OOPSLA*, pages 31–42, 2001.

K. C. Baird. *Ruby by Example*. No Starch Press, San Francisco, CA, USA, 2007. ISBN 9781593271480.

D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

D. Batory. On the importance and challenges of FOSD. In *FOSD '09: Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 1–1, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-567-3.

D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5.

D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004. ISSN 0098-5589.

D. S. Batory and E. Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. UCS*, 14(12):2059–2082, 2008.

M. Becker. Towards a General Model of Variability in Product Families. In *Software Variability Management Workshop*, pages 19–27, Feb. 2003.

K. Berg, J. Bishop, and D. Muthig. Tracing Software Product Line Variability: from Problem to Solution Space. In *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 182–191, , Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists. ISBN 1-59593-258-5.

D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program*, 53(3):333–352, 2004.

D. Bjørner. *Software Engineering, vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science. Springer, 2006.

E. Börger and D. S. Batory. Coupling Design and Verification in Software Product Lines. In S. Hartmann and G. Kern-Isberner, editors, *Foundations of Information and Knowledge Systems, 5th International Symposium, FoIKS 2008, Pisa, Italy, February 11-15, 2008, Proceedings*, volume 4932 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2008. ISBN 978-3-540-77683-3.

J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000. ISBN 0-201-67494-7.

Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen, January 2010. Acceptance rate: 54

N. E. Boustani and J. Hage. Improving Type Error Messages for Generic Java. In G. Puebla and G. Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 131–140. ACM, 2009. ISBN 978-1-60558-327-3.

M. Bravenboer and E. Visser. Concrete Syntax for Objects. Domain-Specific Language Embedding and Assimilation without Restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, Canada, Oct. 2004. ACM Press.

F. P. Brooks Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, Apr. 1987.

D. Chelimsky, D. Astels, Z. Dennis, A. Hellesøy, B. Helmkamp, and D. North. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, Oct. 2010. ISBN 978-1-93435-637-1.

K. Chen, W. Zhang, H. Zhao, and H. Mei. An Approach to Constructing Feature Models Based on Requirements Clustering. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 31 – 40, 29 2005.

R. Chitchyan, I. Sommerville, and A. Rashid. A Model for Dynamic Hyperspaces. In *Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD*, 2003.

S. Clarke and R. J. Walker. Mapping Composition Patterns to AspectJ and Hyper/J. In P. Tarr and H. Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a Feature: A Requirements Engineering Perspective. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008. ISBN 978-3-540-78742-6.

P. Clements and L. Northrop. A Framework for Software Product Line Practice. `http://www.sei.cmu.edu/productlines/tools/framework/`, 2000.

P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Aug. 2001.

P. Cointe. Metaclasses are First Class: The ObjVlisp Model. *SIGPLAN Not.*, 22(12):156–162, 1987. ISSN 0362-1340.

W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178, New York, NY, 1991. Springer-Verlag.

J. Coplien, D. Hoffman, and D. M. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

K. Czarnecki. Overview of Generative Software Development. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *UPP*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004. ISBN 3-540-27884-2.

K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

K. Czarnecki and U. W. Eisenecker. Components and Generative Programming. In O. Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 1999. ISBN 3-540-66538-2.

K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proceedings of the International Workshop on Software Factories (OOPSLA'05*, 2005.

K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In D. S. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2002. ISBN 3-540-44284-7.

K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In R. L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004. ISBN 3-540-22918-3.

K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing Cardinality-based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972. ISBN 0-12-200550-3.

A. M. Davis. Fifteen Principles of Software Engineering. *IEEE Software*, 11 (6):94–96, 101, 1994.

B. Delaware, W. Cook, and D. Batory. A Machine-checked Model of Safe Composition. In *Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 31–35. ACM New York, NY, USA, 2009.

A. v. Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982. Reprinted in Dijkstra's *Selected Writings on Computing: A Personal Perspective*, 1982, pp. 60–66.

S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

T. Ekman. A case study of Separation of Concerns in Compiler Construction using JastAdd II. In *Proceedings of the Third AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.

T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In M. Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer, 2004a. ISBN 3-540-22159-X.

T. Ekman and G. Hedin. Reusable language specification modules in JastAdd II. *Proceedings of the Workshop on Evolution and Reuse of Language Specifications for DSLs*, 2004b.

T. Ekman and G. Hedin. Pluggable Checking and Inferencing of Non-null Types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(7), 2007a.

T. Ekman and G. Hedin. The JastAdd system - Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, 2007b.

T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 1–18, 2007c.

E. Ernst. Separation of Concerns. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

E. Ernst. First-Class Object Sets. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008. ISBN 978-3-642-02443-6.

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008. ISBN 9780596516178.

H. Fulton. *The Ruby way: Solutions and Techniques in Ruby Programming*. Addison-Wesley Professional, second edition, 2006. ISBN 0768667208.

K. Gilroy, E. Comer, K. Grau, and P. Merlet. Impact of Domain Analysis on Reuse Methods. Technical Report C04-087LD-0001-00, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ, Nov. 1989.

R. L. Glass. Practical programmer: The Standish report: does it really describe a software crisis? *Communications of the ACM (CACM)*, 49(8):15–16, Aug. 2006. ISSN 0001-0782.

J. Gosling and H. McGilton. *The Java Language Environment. A White Paper*. SUN Developer Network at Oracle, May 1996.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java (TM) Language Specification, Addison-Wesley*. Addison-Wesley Professional, 2005.

O. Gotel and C. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94 –101, 18-22 1994.

J. D. Gould and C. Lewis. Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM (CACM)*, 28(3):300–311, Mar. 1985. ISSN 0001-0782.

T. S. Group. Chaos Report. Technical report, Standish Group International, 2003.

S. Günther and S. Sunkle. Feature-oriented programming with Ruby. In *FOSD '09: Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 11–18, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-567-3.

S. Günther and S. Sunkle. Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming. In *Proceedings of International Workshop on Feature-oriented Software Development (FOSD)*, October 2010.

S. Günther and S. Sunkle. rbfeatures: Feature-oriented programming with ruby. *Science of Computer Programming*, In Press, Corrected Proof:–, 2011. ISSN 0167-6423. doi: DOI:10.1016/j.scico.2010.12.007.

G. Haddad and G. T. Leavens. Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations. Technical Report CS-TR-08-05, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, Apr. 2008.

S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Transactions on Software Engineering*, 41(4):93–95, 2008.

G. Hedin and E. Magnusson. JastAdd: An Aspect-oriented Compiler Construction System. *Sci. Comput. Program.*, 47(1):37–58, 2003. ISSN 0167-6423.

G. Hedin, J. Akesson, and T. Ekman. Extending Languages by Leveraging Compilers - from Modelica to Optimica. *IEEE Software*, 99(PrePrints), 2010. ISSN 0740-7459.

S. Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

R. Hightower, W. Onstine, P. Visan, and D. Payne. *Professional Java tools for extreme programming: Ant, Xdoclet, JUnit, Cactus, and Maven.* Wrox, 2004. ISBN 978-0764556173.

C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Inf.*, 1: 271–281, 1972.

C. A. R. Hoare and J. Misra. Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005. ISBN 978-3-540-69147-1.

C. Hundt, K. Mehner, C. Pfeiffer, and D. Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology (JOT)–Special Issue: TOOLS EUROPE*, 6(9):417–436, 2007.

W. Hürsch and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, Feb.24 1995.

IEEE-1362-1998. IEEE Guide for Information Technology - System Definition - Concept of Operations (ConOps) Document. *IEEE Std 1362-1998*, 1998.

IEEE-1471-2000. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Std 1471-2000*, 2000.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. ISSN 0164-0925.

ISO/IEC. Guide to Software Product Quality Requirements and Evaluation (SQuaRE), 25000-2005. July 2005.

ISO/IEC. Systems and Software Engineering - Vocabulary. *IEEE Unapproved Draft Std P24765-2009*, Sept 2009.

ISO/IEC-26514:2008. IEEE Draft Standard Adoption of ISO/IEC 26514:2008 - Systems and Software Engineering - Requirements for Designers and Developers of User Documentation. *IEEE Unapproved Draft Std P26514/D1*, December 2009.

M. Jaring and J. Bosch. Representing Variability in Software Product Lines: A Case Study. In *Proceedings of the Second International Conference on Software Product Lines*, pages 15–36. Springer-Verlag, 2002. ISBN 3-540-43985-4.

G. F. Johnson and D. Duggan. Stores and Partial Continuations as First-class Objects in a Language and its Environment. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–168, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7.

M. Jørgensen and K. Moløkken-Østvold. How Large Are Software Cost Overruns? A Review of the 1994 CHAOS Report. *Information & Software Technology*, 48(4):297–301, 2006.

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

K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Software Eng*, 5:143–168, 1998.

C. Kästner. *Virtual Separation of Concerns : Preprocessors 2.0.* PhD thesis, School of Computer Science, University of Magdeburg, May 2010.

C. Kästner and S. Apel. Type-checking Software Product Lines - A Formal Approach. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, Sept. 2008. ISBN 978-1-4244-2187-9.

C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, Sept. 2009.

C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.

C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of The Optional Feature Problem: Analysis and Case Studies. In *SPLC '09: Proceedings of the 13th International Software Product Line Conference*, pages 181–190, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005. ISBN 3-540-27992-X.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *European Conference on Model Driven Architecture Foundations and Applications*, pages 331–348. Springer, 2005.

D. E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990. Paris.

P. Lago, H. Muccini, and H. van Vliet. A Scoped Approach to Traceability Management. *Journal of Systems and Software*, 82(1):168 – 182, 2009. ISSN 0164-1212. Special Issue: Software Performance - Modeling and Analysis.

A. Lai, G. C. Murphy, and R. J. Walker. Separating Concerns with HyperJ: An Experience Report. In P. Tarr, A. Finkelstein, W. Harrison, B. Nuseibeh, H. Ossher, and D. Perry, editors, *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, June 2000.

G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for Enhanced Languages and Methods to Aid Verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006. ACM. ISBN 1-59593-237-2.

S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 33(10) of *ACM SIGPLAN Notices*, pages 36–44, New York, NY, Oct. 1998. ACM, ACM.

T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. ISBN 0-201-43294-3.

J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *ICSE*, pages 112–121, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1.

R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In A. P. Black, editor, *ECOOP*

2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005a. ISBN 3-540-27992-X.

R. E. Lopez-Herrejon. The Expression Problem as Product-Line and its Implementation in AHEAD. Technical report, Department of Computer Sciences, University of Texas at Austin, October 2004.

R. E. Lopez-Herrejon and D. S. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 10–24, London, UK, 2001. Springer-Verlag. ISBN 3-540-42546-2.

R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. Extended Report. Technical Report CS-TR-05-16, The University of Texas at Austin, Department of Computer Sciences, 1 2005b. Fri, 28 Sep 107 13:03:38 GMT.

N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language Support for Managing Variability in Architectural Models. In *Software Composition*, pages 36–51, 2008.

E. Magnusson and G. Hedin. Circular Reference Attributed Grammars — Their Evaluation and Applications. *Sci. Comput. Program.*, 68(1):21–37, 2007. ISSN 0167-6423.

D. Malayeri. CZ: Multiple Inheritance without Diamonds. In G. E. Harris, editor, *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, pages 923–924. ACM, 2008. ISBN 978-1-60558-220-7.

M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 127–136, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

M. Matinlassi, E. Niemelä, and L. Dobrica. Quality-Driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture. Technical Report VTT-PUBS-456, VTT Electronics, Jan. 2002.

Y. Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001. ISBN 0596002149.

S. McConnell. From The Editor - Software Engineering Principles. *IEEE Software*, 16(2), 1999.

S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.

J. D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, Dec. 2001.

M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 127–136. ACM Press, 2004.

M. Moodie. *Pro Apache Ant*. Springer, Dordrecht, 2006. ISBN 978-1-590595596.

P. Mulet, J. Malenfant, and P. Cointe. Towards a Methodology for Explicit Composition of MetaObjects. *ACM SIGPLAN Notices*, 30(10):316–330, Oct. 1995. ISSN 0362-1340.

P. Naur and B. Randell, editors. *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, Jan. 1969. NATO Scientific Affairs Division.

O. Nierstrasz and F. Achermann. Separation of Concerns through Unification of Concepts. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

E. Nilsson-Nyman, T. Ekman, G. Hedin, and E. Magnusson. Declarative Intraprocedural Flow Analysis of Java Source Code. In *Proceedings of 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, 2008.

N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction: 12'th International Conference, CC 2003*, volume 2622, pages 138–152, New York, NY, Apr. 2003. Springer-Verlag.

H. Obbink, R. van Ommering, J. G. Wijnstra, and P. America. Component-Oriented Platform Architectures For Software Intensive Product Families. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 99–141. Kluwer Academic Publishers, 2002.

H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art InSoftware Development*. Kluwer, 2000a.

H. Ossher and P. L. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *ICSE*, pages 734–737, 2000b.

D. L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, 1972a.

D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)*, 15(12):1053–8, Dec. 1972b.

D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, Mar. 1979.

D. L. Parnas, P. C. Clements, and D. M. Weiss. The Modular Structure of Complex Systems. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 408–417, Piscataway, NJ, USA, 1984. IEEE Press. ISBN 0-8186-0528-6.

P. Perrotta. *Metaprogramming Ruby*. Pragmatic Bookshelf, Feb 2010. ISBN 978-1-93435-647-0.

K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin Heidelberg New York, 2005. ISBN 3-540-24372-0.

C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443, 1997.

R. Prieto-Diaz. Domain Analysis For Reusability. In *Proceedings of the Eleventh Annual International Computer Software and Application Conference*, pages 63–69, 1987. The focus is on the DA process, NOT on reuse.

M. Pukall, C. Kästner, and G. Saake. Towards Unanticipated Runtime Adaptation of Java Applications. In *APSEC*, pages 85–92. IEEE, 2008. ISBN 978-0-7695-3446-6.

M. Pukall, C. Kästner, S. Götz, W. Cazzola, and G. Saake. Flexible Runtime Program Adaptations in Java - A Comparision. Technical Report FIN-014-2009, Department of Computer Science, Otto-von-Guericke University of Magdeburg, Germany, Nov. 2009.

M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA, June 2002.

M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, Jan. 2010.

M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. ur Rahman. Modeling Dependent Software Product Lines. In *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 13–18. Department of Informatics and Mathematics, University of Passau, Oct. 2008a.

M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008b.

M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Combining Static and Dynamic Feature Binding in Software Product Lines. Technical Report 13, Fakultät für Informatik, Universität Magdeburg, Sept. 2009.

RSpec. Behaviour Driven Development for Ruby. `http://rspec.info/`, 2010.

Ruby Shoes Development Community at GitHub. Shoes, a Tiny Toolkit. `http://github.com/shoes/shoes/wiki`, 2009.

RubyHome. Ruby Language Homepage. `http://www.ruby-lang.org/en/`, September 2010.

M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 277–294, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3.

M. Schäfer, M. Verbaere, T. Ekman, and O. Moor. Stepping Stones over the Refactoring Rubicon. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3.

N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. *Lecture notes in computer science*, pages 248–274, 2003.

M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D. A. Lamb, editor, *ICSE Workshop on Studies of Software Design*, volume 1078 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1993. ISBN 3-540-61285-8.

M. Sinnema and S. Deelstra. Classifying Variability Modeling Techniques. *Information & Software Technology*, 49(7):717–739, 2007.

M. Sinnema, O. de Graaf, and J. Bosch. Tool Support for COVAMOF. In *Workshop on Software Variability Management for Product Derivation*, Boston, MA, Aug. 2004a.

M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In R. L. Nord, editor, *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2004b. ISBN 3-540-22918-3.

P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In *ECBS*, pages 308–318. IEEE Computer Society, 2006. ISBN 0-7695-2546-6.

I. Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004. ISBN 0321210263.

H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.

StandishGroupReport. The CHAOS Report. `http://www.standishgroup.com/sample_research/chaos_1994_1.php`, 1994.

W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

B. Stewart. An Interview with the Creator of Ruby. `http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`, November 2001.

C. Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, 2000. ISSN 1388-3690.

R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: Core Design and Semantic Definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

S. Sunkle. Feature-oriented Decomposition of SQL:2003. Master's thesis, Department of Computer Science, University of Magdeburg, Germany, October 2007.

S. Sunkle and M. Pukall. Using Reified Contextual Information for Safe Runtime Adaptation of Software Product Lines. *7th ECOOP'2010 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*, June 2010.

S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, and G. Saake. Generating highly customizable sql parsers. In *SETMDM '08: Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, pages 29–33, New York, NY, USA, 2008a. ACM. ISBN 978-1-59593-964-7.

S. Sunkle, M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and G. Saake. Features as First-class Entities – Toward a Better Representation of Features. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 27–34. Department of Informatics and Mathematics, University of Passau, Oct. 2008b.

S. Sunkle, S. Günther, and G. Saake. Representing and Composing First-class Features with FeatureJ. Technical Report FIN-017-2009, Department of Computer Science, Otto-von-Guericke University of Magdeburg, Germany, Nov. 2009.

P. Systems. pure::variants Connector for IBM Rational DOORS. `http://www.pure-systems.com/DOORS.102+M54a708de802.0.html`, 2010.

C. A. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison Wesley, second edition edition, 2002. ISBN 0-201-74572-0.

P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, pages 107–119, 1999.

M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for java. In *Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, Denver, Colorado, USA, 2000. ISBN 3-540-67761-5.

S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007. ISBN 978-1-59593-855-8.

D. Thomas and A. Hunt. *Programming Ruby: the Pragmatic Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0-201-71089-7.

F. Van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action : The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working Conference on Software Architecture (WICSA)*, pages 45–55. IEEE Computer Society, 2001.

B. Venners. The Philosophy of Ruby - A Conversation with Yukihiro Matsumoto, Part I. `http://www.artima.com/intv/ruby.html`, September 2003a.

B. Venners. Blocks and Closures in Ruby - A Conversation with Yukihiro Matsumoto, Part III. `http://www.artima.com/intv/closures.html`, December 2003b.

B. Venners. Dynamic Productivity with Ruby - A Conversation with Yukihiro Matsumoto, Part II. `http://www.artima.com/intv/tuesday.html`, November 2003c.

P. Wadler. The Expression Problem in the Email to the Java Genericity mailing list, Dec. 1998. URL `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`. Email to the Java Genericity mailing list.

D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999. ISBN 0-201-69438-7.

N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, Apr. 1971. ISSN 0001-0782.

N. Wirth. From Programming Language Design to Computer Construction. *Commun. ACM*, 28(2):160–164, 1985. ISSN 0001-0782.

F. Xie and J. C. Browne. Verified Systems by Composition from Verified Components. In *ESEC / SIGSOFT FSE*, pages 277–286. ACM, 2003.

B. Xin, S. McDirmid, E. Eide, and W. C. Hsieh. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, The University of Utah, 2004.

E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979. ISBN 0138544719.

P. Zave. An Experiment in Feature Engineering. pages 353–377, 2003.

M. Zenger and M. Odersky. Implementing Extensible Compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, June 2001.

S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza. VML* - A Family of Languages for Variability Management in Software Product Lines. In M. van den Brand, D. Gasevic, and J. Gray, editors, *SLE*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2009. ISBN 978-3-642-12106-7.