The Dissertation Committee for Roberto Erick Lopez Herrejon
certifies that this is the approved version of the following dissertation:

Understanding Feature Modularity

Committee:

Don Batory, Supervisor

William Cook

James C. Browne

Dewayne Perry

Oege de Moor

Paul C. Clements

Understanding Feature Modularity

by

Roberto Erick Lopez Herrejon, B.Eng., M.S.C.S, M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

To my family

Acknowledgments

Like any worthy human endeavor, pursuing a PhD involves many people over many years in many roles. First I would like to thank my mother for her, although unconventional, extensive support, love, and care during all my life. Her tenacity and perseverance are an example for me. My brother Valdemar, my sister Reyna, my uncle Benjamin and aunt Ernestina have supported me in so many ways throughout my life. Gracias.

Many teachers, mentors, and professors have influenced and shaped my academic and personal life in so many ways. Among them the late Julius Schlusche taught me English and other languages, but most importantly how to read between the lines and look at life and things from different perspectives. I really miss our Saturday afternoon sessions with the follow-up snacks and conversations with his wife Teresa Perdomo. At UNAM, Elisa Viso and Hanna Oktaba guided me and helped me prepare for my doctoral studies.

During these many years I had the fortune to meet friends and colleagues that enriched my stay in Austin: Joohyung Lee and his wife Jeehyun Park, Madhu and Rachna Kayastha, Axel Rauschmayer, Jia Liu, Jacob Neal Sarvela, Yancong Zhou, and Claudia Torres-Garibay. The entire staff of the Computer Sciences Department made me feel at home from the beginning. Gloria Ramirez and Katherine Utz pampered us with candies, chocolates, and Christmas presents and made the sometimes painful administrative stuff run smoothly.

In every dissertation there is usually an acknowledgement for the advisor. For me this is a very difficult task. What can I say about my advisor Don Batory in such a short space?. I can simply say that it has been an honor and a true pleasure working with him. His support, patience, encouragement, advice, and overall help has gone far and beyond the call of duty. His passion, professionalism, work ethics, strive for excellence, and human quality are an example for me to follow. Thank you Don.

I thank my committee members William Cook, James Browne, Dewayne Perry, Paul Clements, and Oege de Moor for their valuable feedback.

My colleagues and supervisors at Oxford provided a supportive and flexible working environment for me to finish my research and dissertation.

Last but not least, my deepest thanks go to my wife Lupita. She endured, at least as much as I did, all the ups and downs of graduate life and dissertation writing. Throughout the ordeal she always helped me put things in a human perspective, kept my feet on the ground and a smile on my face. Gracias. Te amo Pecas.

ROBERTO ERICK LOPEZ HERREJON

Oxford, England
June 2006

Understanding Feature Modularity

Publication No. _____

Roberto Erick Lopez Herrejon, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Don Batory

Features are increments in program functionality. Feature abstraction, the process of abstracting programs into their constituent features, is a relatively common yet informal practice in software design. It is common because it simplifies program understanding. It is also important for software product lines whose essence is the systematic and efficient creation of software products from a shared set of assets or features, where each product exhibits common functionality with other products but also has unique functionalities.

Thus, it seems natural to modularize feature abstractions and use such modules as building blocks of programs and product lines. Unfortunately, conventional modularization approaches such as methods, classes and packages are not geared for supporting feature modules. They present two recurrent problems. First, a typical feature implementation is spread over several conventional modules. Second, features are usually more than source code artifacts as they can modularize many different program representations (makefiles, documentation, performance models).

An undesirable consequence is that developers must lower their abstractions from features to those provided by the underlying implementation languages, a process that is far from simple let alone amenable to significant automation. The conceptual gap created between feature abstractions and their modularization hinders program understanding and product line development. The root of the problem is the fact that feature modularity is not well understood and thus not well supported in conventional programming languages, modularization mechanisms, and design techniques.

In this dissertation, we explore language and modularity support for features founded on an algebraic model geared for program synthesis. Our model integrates ideas from collaboration-based designs, mixin layers, aspect oriented programming, multi-dimensional separation of concerns, and generative programming. We assess our model with an implementation of a non-trivial product line case study, and evaluate feature support in emerging modularization technologies.

Contents

Chapter 1

Introduction

1.1 Overview and Contribution

Program abstraction and modularity are two of the driving factors behind the evolution of software development. Abstraction allows us to focus on the relevant parts of a problem while modularity lets us decompose problems and compose solutions in manageable units to cope with program complexity.

The history of programming languages and their attendant software development methodologies has been that of continuously raising the abstraction level from bits and bytes towards concepts and language constructs that resemble more and more the abstractions of problem domains, and supporting the corresponding abstractions with adequate modularity mechanisms. Examples are procedures or methods that led to Structural Programming, and classes and objects that are the focus of Object Oriented Programming.

We propose features for program abstraction and modularization to address documented shortcomings of objects and classes, the current leading paradigm for software development. We provide an algebraic model, and its underpinning implementation support, of feature modules that describes the analysis and synthesis of programs as mathematical expressions subject to algebraic laws. We employ algebraic models because they abstract architectural details from tool-specific implementation details. Representing programs as algebraic expressions of feature modules shifts our focus from implementation details to architectural reasoning: what features constitute a program? and what is their architecture (denoted by algebraic expressions that compose features)?. Algebraic models

also facilitate the application of automatic and generative techniques of program synthesis and product line generation.

Informally, a feature is a prominent or salient part of an object or thing. Every day objects like cars, houses, or dogs are distinguished among similar objects by the set of features they exhibit such as color, size, or breed. A similar scenario can be applied to software programs where features correspond to the functionality that programs provide. For instance, a word processor has features of file loading and saving, editing options, spell checking, font formatting, printing options, and so on. Feature abstraction is a relatively common yet informal practice in software design, as exemplified by UML use cases [125], because it simplifies program understanding. Feature abstraction is also important for software product lines whose essence is the systematic and efficient creation of software products from a shared set of assets or features, where each product exhibits common functionality with other products but also has unique functionalities.

It seems natural then to modularize features and use such modules as building blocks of programs and product lines. Unfortunately, conventional modularization approaches such as methods, classes or packages are not geared for feature modules and present two recurrent problems. First, a typical feature implementation is spread over several conventional modules. Second, features are usually more than source code artifacts as they can modularize many different program representations such as makefiles, documentation, or performance models.

As a negative consequence, software developers must lower their abstractions from features to those provided by the underlying implementation languages, a process that is far from simple let alone amenable to significant automatization. The conceptual gap created between feature abstractions and their modularization hinders program understanding and product line development. The root of the problem is the fact that feature modularity is not well understood and thus not well supported in conventional programming languages, modularization mechanisms, and design techniques.

The main contributions of this dissertation are:

- A foundation of *Feature Oriented Programming (FOP)*. FOP is a software development paradigm that focuses on feature modularity. It generalizes and builds on the success of *Relational Query Optimization (RQO)*. The key insight derived from RQO is expressing programs and their designs in terms of relational algebra expressions whose operations are composed and possess mathematical properties. Such program specifications are amenable to automatic programming (derivation of optimized programs from unoptimized ones), and generative programming (generation of efficient code from higher-level specifications). FOP raises the level of abstraction from low-level module implementations to mathematical entities that represent features and are subject to algebraic laws. FOP aims at providing a general theory of software development in the same sense that relational algebra is a theory upon which query evaluation programs that access relational databases are built.
- An evaluation of feature modularity in novel modularization technologies. In recent years, there has been an increasing interest in the study of new modularization techniques to help overcome limitations of conventional object oriented programming languages. The result has been a plethora of tools and techniques[1]. Such diversity makes difficult the comparison among the different approaches and an assessment of how they could be applied to product lines. To address this problem, we developed a canonical example of product lines and proposed basic properties that feature modules that implement this example should provide [106]. We evaluated these properties in several technologies to assess their support for feature modularity in product line development and relate these properties to an algebraic model, thus providing an implementation-independent model on which to compare and contrast modularization technologies.
- An algebraic model of a core of AspectJ. *Aspect Oriented Programming (AOP)* is a popular emerging modularization technology [93]. Its flagship tool is AspectJ, an extension of Java whose goal is to modularize aspects, modules that implement

1. For instance, [15] surveys 27 approaches.

crosscutting concerns, that is, their implementation cuts across other module boundaries (such as classes) or it is intertwined with fragments of other concerns. This model helped us uncover some of the complexity in using aspects in product lines and incremental development.

- A functional model of aspect composition. We propose an alternative model that solves the problems encountered with the defacto model of AspectJ while leaving AspectJ's power intact. It merges contributions from FOP and AOP, and helps contrast and compare the differences of both paradigms, a common source of controversy, misconceptions and confusion in the software engineering community.
- An empirical assessment of the functional model of aspects. We implemented a product line of non-trivial size in AspectJ to assess the validity and relevance of our model. We emulated functional composition by preventing the problems identified with the defacto AspectJ model and collected statistics that shed light on the impact different constructs have on feature modules and on the overall structure of the product line.

1.2 Outline

The subsequent chapters are structured as follows:

In Chapter 2 we put our work in context. First, we provide working definitions of features, modules, and product lines that we use to define feature modularity. Second, we describe the two main processes recurrent in many product line methodologies, Domain Engineering and Application Engineering, and set our work within the context of the main activities of these two processes. Third, we present a classification of techniques for variability management (dealing with product variations) and categorized our work within that classification. We finish the chapter by relating our work with research in software architecture and program synthesis.

In Chapter 3 we lay out the foundations of FOP. GenVoca [29], a methodology for creating application families and architecturally extensible software, was the first model

of FOP. GenVoca was extended by AHEAD [1][30] whose feature modules are hierarchical and can include other artifacts besides source code. In this chapter, we describe an algebraic representation of AHEAD's feature modules, their composition, and implementation.

In Chapter 4 we propose a canonical example of product lines and a list of basic properties of features modules. We perform an evaluation of feature modularity in several novel modularization technologies with this example and properties.

In Chapter 5 we develop an algebraic model of a core of AspectJ. We analyze and illustrate with this model some of the problems associated with aspects in the implementation of product lines and in incremental development. We propose an alternative model of composition, the functional model, to address these problems. We compare and contrast the functional model with the model of FOP explained in Chapter 3, and relate it to the feature module properties presented in Chapter 4.

Chapter 6 presents an empirical assessment on the validity and relevance of our functional model of aspects. We implemented a product line of non-trivial size emulating functional composition in AspectJ and collected statistics on the use of language constructs. We analyze the statistics collected in relation to different types of features we identified and the relative strengths of FOP and AOP.

In Chapter 7 we summarize our results and contributions and discuss several venues for future research.

Chapter 2

Features, Modules, and Product Lines

In this chapter we put our work and contributions in the context of program modularity and research on software product lines, software architecture and program synthesis.

Features, modules, and software product lines are concepts whose definitions are overloaded or not generally agreed upon. We illustrate the diversity of definitions with samples extracted from research literature. Within this diversity, we frame our working definitions for these concepts. We also introduce and define the concept of feature module that we elaborate and illustrate in subsequent chapters.

Over the last decade, many product line methodologies have been proposed with different goals, processes, artifacts, and perspectives. Despite the significant disparity, two common threads can be distinguished: a) the distinction between two processes, Domain Engineering and Application Engineering, and b) the importance of identifying and handling the common and different properties of member programs, usually referred to as *commonality* and *variability* management. We describe how our work fits in the basic activities of these two processes and how feature modules can be classified within the different types of variability management mechanisms.

Finally, we relate our work with research in software architecture and program synthesis.

2.1 Features

Informally, a *feature* is a characteristic that is prominent in an entity or thing. Unfortunately, in the Software Engineering literature there is no general agreement on what a feature is, let alone on its properties and implications. Some of the definitions are:

- End-user visible characteristics of a system [85].
- A distinguishable characteristic of a concept that is relevant to some stake-holder [85].
- Qualitative properties of concepts [51].
- A functional requirement; a reusable product line requirement or characteristic. A requirement or characteristic that is provided by one or more members of a software product line [68].
- Increment in program functionality [162].

All these notions of features are valid and more relevant for different stages of product line development or for different product line methodologies. For our work we focus on the last definition, thus throughout the dissertation we regard features as increments in program functionality.

Software systems are designed and built to meet the requirements imposed by users, customers, markets, etc. Broadly speaking, requirements can be categorized into [67]:

- Functional requirements: Describe what a system does using informal or formal notations.
- Non-functional requirements: Describe how a system meets the functional requirements and have a strong relation to the system's architecture. Common non-functional requirements are: availability, security, safety, integrity, and speed. Non-functional requirements are also known as *quality attributes* [42].

Since we regard features as increments in program functionality, our work mainly addresses functional requirements. Nevertheless, in Chapter 3 we describe how we deal with artifacts associated with quality attributes such as performance models.

2.2 Modules

The importance of building software systems modularly was recognized in the early days of Software Engineering [133]. Among other benefits, modularity makes systems easier to understand, evolve, use, and scale [67][118][122]. Despite its significance, there is no universal consensus as to what constitutes a software module let alone the properties it should exhibit. One of the most thorough analysis of modularity, its properties and Software Engineering implications has been presented by Bertrand Meyer in his seminal book on Object Oriented Software Construction [118]. He describes five criteria that software methods should have to be considered modular, five rules to follow to ensure modularity, and five principles derived from the criteria and rules.

Five Criteria. The methodology should provide mechanisms to decompose systems into smaller and simpler subsystems (*decomposability*), and support composition of complex systems from simpler ones (*composability*). It should allow developers to understand each module without having to inspect all other modules that constitute a system (*understandability*). It should limit the impact of specification changes to one or a small number of modules (*continuity*) and contain run-time errors within the module or neighboring modules where the error occurred (*protection*).

Five Rules. The modules identified at the design level should correspond to modules at the implementation level (*direct mapping*). Every module should communicate with the fewest number of other modules (*few interfaces*), exchanging as little information as possible (*small interfaces*) in a clear and explicit manner (*explicit interfaces*), and exposing only a subset of its properties to other modules (*information hiding*).

Five Principles. Modules should correspond to syntactic units in the language (*linguistic modular units*), contain all information of a module within the module itself (*self-documentation*), provide a uniform notation to its services whether stored or computed (*uniform access*), and be open for extension and closed when available for use by other modules (*open-close*). Additionally, when the system can support a set of alternatives their definitions should be confined to a single module (*single choice*).

All these facets of modularity are indeed important; however, our focus is on:

- **Criteria**. Decomposability and Composability because we want to decompose and compose systems by features.
- **Rules**. Direct mapping because we want our modules to reflect a feature-level design.
- **Principles**. Linguistic modular units because we want to have language constructs to express features, self-documentation because we expect our modules to not only contain source code but also other artifacts, open-close because we want our modules to be extensible yet usable by other modules.

Thus, in the context of Meyer's modularity framework and our focus within it, our working definition of feature module is:

A feature module is an extensible modularization mechanism supported by language constructs and implementation machinery for decomposing and composing systems in terms of features that can be realized with multiple artifacts.

Let us dissect this definition and relate it to the criteria, rules and principles that we focus on:

- A feature module is an *extensible modularization mechanism* because we want our modules to meet the open-close principle, being extensible yet usable by other modules.
- A feature module is *supported by language constructs and implementation machinery*, this relates to the principle of linguistic modular units.
- Support *for decomposing and composing and composing systems*, relates to the criteria of decomposability and composability.
- The fact that composition and decomposition is *in terms of features* relates to the rule of direct mapping as we want to reflect our feature-level designs as modules in our implementation.
- We state that feature modules *can be realized with multiple artifacts* to establish a relation with the self-documentation principle.

Throughout the dissertation we elaborate more on and illustrate this definition.

2.3 Product Lines

Software product lines or product families were also conceived in the early days of Software Engineering [134]. Like features and modules, there is no general agreement on what is a software product line and how to model, design and build them. The following is a sample of definitions of product line:

- A set of programs studied in terms of the common and special properties of the individual members [134].
- A set of software-intensive systems sharing a common, managed set of features that satisfy 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 [42].
- A group of products sharing a common managed set of features that satisfy the specific needs of a selected market [160].
- A family of products designed to take advantage of the common aspects and predicted variabilities [158].
- A family of software systems that have some common functionality and some variable functionality [68].

These definitions emphasize the perspectives taken by the different product line methodologies. There is a common thread in them however, the recognition that products in a product line share a common set of assets or features, and present differences among them. These two facts are referred as *commonality* and *variability* respectively.

Our work acknowledges the importance of these two concepts while highlighting that commonalities and variations within a product line are based on the features that member programs implement. Thus our working definition is:

A software product line is a set of similar programs that are distinguished by the set of features they implement[1].

It is important to reiterate at this point that our working definitions of features, modules, and product lines fall within the range of definitions of current research litera-

1. Typically features are shared among several programs thus constituting a common core of a product line.

ture. This not only sets our work within a research context but also allowed us to build a plausible working definition of feature modules.

2.4 Product Line Methodologies

Many methodologies have been developed to create product line architectures, some examples are: Feature-Oriented Domain Analysis (FODA) [85], Feature Oriented Reuse Model (FORM) [86], Family-oriented Abstraction Specification and Translation (FAST) [158], FeatuRSEB [71], SEI Framework for Product Line Practice [39][42], ProdUct Line Software Engineering (PuLSE) [32], Evolutionary Software Product Line Engineering Process (ESPLEP) [68], Quality-driven Architecture Design and quality Analysis (QADA) [111], Product Line Use case modeling for Systems and Software engineering (PLUSS) [62], Kobra [16], PRIME [136][2].

Despite the enormous disparity among methodologies regarding the number of processes, activities, tasks, artifacts or perspectives; a common theme can be identified. Product line methodologies broadly categorize their processes in two:

- *Domain Engineering* is the process that analyzes, designs, and implements the commonality and variability of a product line.
- *Application Engineering* is the process that captures product requirements, creates a map (configuration) from requirements to common and variable assets obtained in the Domain Engineering process, and builds (generates) the software products.

These two processes are depicted in Figure 2.1. We omit the lines in the figure as we do not focus on the order in which these activities can be performed. Within these two process our work fits as follows:

- Domain engineering: our work touches on design but fundamentally addresses implementation issues.

2. More approaches appear in [51] and [136].

- Application engineering: our work touches on configuration issues but concentrates on the generation of product variants.

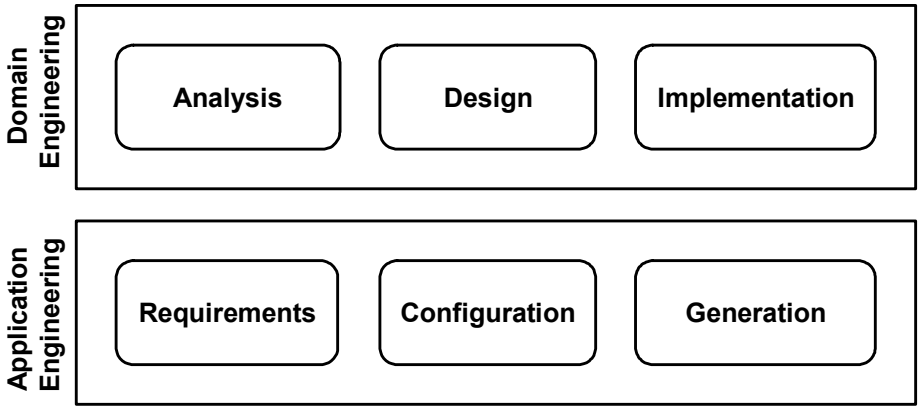


Figure 2.1 Overview of Domain and Application Engineering

2.5 Variability Management

Variability is a key and pervasive concept in product line engineering. A consequence of the proliferation of product line methodologies is a large range of mechanisms to express and implement variability. Next we present a classification of variability mechanisms and describe where our work fits in this classification [6][42][51][80].

Language Supported. Rely on the language philosophy, constructs, and implementation mechanisms.

- Inheritance mechanisms such as single, multiple, and mixin inheritance support variability by adding functionality to classes [38][122][147].
- Generic programming abstracts variability into generic parameter types [51].
- Reflection and Metaprogramming permit variation in program execution based on the program state or its representation [6][51].

Compiler or Linker Supported. Based on compiler or linker technology.

- Libraries are collections of functions or components that can vary for different combinations of features.

- Classloaders support variability by loading different versions of classes depending on the program execution or configuration [6].
- Preprocessors support variability by including or excluding code fragments or programs before compilation.

Frame Technology. A *frame* is a text template that contains commands and code fragments [20][76]. These commands generate code particular to a variant, usually according to one or more parameters, in a process called frame instantiation. An example of frame-based technology for handling variants in software product lines is XVCL (XML-based Variant Configuration Language) [164].

Design Patterns. Programming protocols for common and recurrent functionality. They support variability as they can be adapted for different sets of classes and usage scenarios [66].

Frameworks. A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software [66]. Variability is supported by extending abstract classes of the framework to provide them with the concrete implementation of a product variant or *frame instance* [23].

Modularization Technologies. Recently, several technologies have been proposed that address modularization shortcomings of current Object Oriented programming languages and technologies. Some of them have already been used to implement product lines [104][151][161].

Our work falls in the last category because we propose feature modules as a new modularization technology conceived to support implementation and synthesis of product line designs. In Chapter 4 we illustrate the use of several modularization technologies for the implementation of a product line example.

2.6 Software Architecture and Program Synthesis

Despite being commonly used in current day practice and research literature, there is no common and universally accepted definition of the term *Software Architecture*. We use Shaw and Garlan's as our working definition [146]:

Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Extensive research has been carried out in software architecture since the mid 1980s. Researchers soon realized that there was a need to develop formal notations and tool support to describe, analyze, and implement architectural specifications of systems. This realization originated *Architecture Description Languages (ADLs)*, which broadly speaking provide syntactic and semantic models to describe systems at a higher-level view not at implementation details. A vast array of ADLs have been proposed to address different problem domains, expressed in different languages and supported by different semantic models [117]. Examples of ADLs are: *Weaves* for parallel processing whose component units are *tool fragments* (procedures) connected via ports that communicate by passing objects [69], *Darwin* which is a declarative language (formally defined via π-calculus) for the hierarchical composition of distributed systems built with components interacting via services [109], and *Wright* which specifies and analyzes software architectures in terms of components (ports and specification) and connector types theoretically underpinned in CSP [3].

Other component-based programming examples for parallel and distributed systems are: *SBASCO* which merges *skeletons* (code patterns) and components to provide high-level programmability and performance portability [52], *ASSIST* which also uses skeletons and components but its applications are specified in a coordination language [154], *Common Component Architecture (CCA)* which is a large effort to define standards for components for high-performance computation [9], and *P-COM*^{2} that targets develop-

ment of program families from components encapsulated with interfaces that describe their properties which are used in the composition to produce near-optimal results for specific applications and execution environments [110]. These approaches are not typically described as ADLs, nonetheless they pursue the same goal of describing system architecture and composition with syntactic and semantic models that abstract implementation details.

Along the same lines, our work could be viewed as an example of ADLs that uses an algebraic notation to specify how components (features) of systems are composed. However, contrary to most ADLs, we do not focus on particular application domains. We believe our work could potentially be used in conjunction with existing ADLs because we regard features as increments in program functionality, view that fits the composition units of ADLs, and the different program specifications used by ADLs can be regarded as program artifacts that could be composed as described in Chapter 3.

Our work can also be put within the context of *Program Synthesis* whose goal is to mechanically synthesize (generate) correct and efficient code from declarative specifications [91]. Kreitz divides program synthesis into three categories [91]:

- *Proofs as programs* regards program synthesis as a proof process. It relies on standard theorem proving techniques.
- *Transformational* program synthesis typically uses rewriting techniques to transform a specification into a form that can be translated to a program in a straightforward manner.
- *Knowledge based* program synthesis generates programs by developing strategies to derive the parameters required to instantiate algorithmic schemas.

Our work falls into the transformational approach because we regard features as program transformations that are mathematically modeled as functions from programs to programs as will be explained in Chapter 3.

2.7 Summary

In this chapter we put our work within the context of modularity, product line and software architecture research, and program synthesis. First we presented our working definitions of features, feature modules and software product lines, and noted that they derive from commonly accepted definitions in current research literature.

Second, we described how our work fits in the product line development process: in the realm of Domain Engineering it addresses design and implementation activities, while in the Application Engineering realm it touches on configuration but focuses mostly on product generation.

Third, we saw that variability management, expressing and implementing product variations, is fundamental for product line engineering. We presented a classification of variability management mechanisms and classified our work as a modularization technology because we aim at providing language level and implementation mechanism support for feature modules.

Finally, we described how our work could be considered as an example of Architecture Description Languages and why we believe it could potentially be used with other examples of ADLs. We classified our approach to program synthesis as transformational as we regard features as program transformations as we will describe in next chapter.

Chapter 3

Feature Oriented Programming

In Chapter 2 we identified feature modules as the building blocks of software product lines. In this chapter, we describe *Feature Oriented Programming (FOP)*, a technology that studies feature modularity and its use in program synthesis. We explain the implementation, algebraic representation and composition of features modules in FOP[1]. In Chapter 4 we present an evaluation of how other modularization technologies could be used to implement features modules.

3.1 Basic Ideas of FOP – GenVoca

FOP has its roots in *GenVoca*, a methodology for creating application families and architecturally extensible software [29]. In this section we explain the basic ideas behind GenVoca, and in next section we present their generalization in AHEAD.

Let us start by considering a standard Java class like:

```
class Lit {
  int value;
  Lit (int v) { value = v; }
  void print() { System.out.print(value);  }
  int eval() { return value; }
}                                                    (1)
```

1. The core of this chapter is based on our ASE paper [27].

This class can be defined in several ways using inheritance, for instance[2]:

```
class Litc {
  int value;
  Lit (int v) { value = v; }
}
class Litp extends Litc {
  void print() { System.out.print(value);  }
}
class Lite extends Litp {
  int eval() { return value; }
}
class Lit extends Lit3 { }                                    (2)
```

The two definitions of `Lit`, in terms of their functionality[3], are identical. They both provide the same field, constructor, and methods `print` and `eval`[4]. However there is one restriction, the order in which inheritance is done matters. In this example, class `Litc` should be at the root of the class hierarchy because it defines field `value` which is referred to by classes `Litp` and `Lite`. Note however, that the order of these last two classes can be interchanged and still yields the same functionality:

```
class Lite extends Litc {
  int eval() { return value; }
}
class Litp extends Lite {
  void print() { System.out.print(value);  }
}
class Lit extends Litp { }                                    (3)
```

2. Suffixes c, p, and e in the class names stand for the functionality they implement core, print, and eval.
3. Remember from Chapter 2 that we do not address information hiding issues (thus we do not consider modifiers such as `public`, `private`, etc.), and we regard features as increments in program functionality.
4. The examples we develop in this chapter are variations of the Expression Problem. We shortly describe this problem in Section 3.4.2, and explain it in more detail on next chapter where we use it as the basis for an evaluation of several modularization technologies.

Another way to think about inheritance is as an operation that increments or refines the functionality of a class. From this perspective, class `Lit` can be expressed as follows:

```
class Lit {                    // Litc
  int value;
  Lit (int v) { value = v; }
}                                                              (4)

refines class Lit {            // Litp
  void print() { System.out.print(value);  }
}                                                              (5)

refines class Lit {            // Lite
  int eval() { return value; }
}                                                              (6)
```

where `refines` is a modifier keyword that indicates a *class refinement* or *class extension*. Note that the name of the class and the two class refinements is `Lit`, thus throughout the chapter we use subscripts (in substitution of suffixes) to distinguish classes and their different refinements.

Classes and their refinements can be expressed algebraically by using functions. We distinguish a special kind of functions, those that receive no arguments and always return the same constant value. We refer to this kind of functions as *values* (a.k.a. constant functions) and to any other types of functions simply as *functions*.

For example, class `Lit` in (4) is a value – a basic source code artifact (in this case a standard Java class), and can be denoted with term $\texttt{Lit}_\texttt{c}$ (subscript `c` refers to the core definition in (4)). Note that parenthesis are omitted in the case of value terms. A class refinement adds functionality to a class to create an extended or refined class. Thus class refinements can be regarded as functions that maps programs (classes) to programs (classes). For instance, the class refinements in (5) and (6) denoted with $\texttt{Lit}_\texttt{p}$ and $\texttt{Lit}_\texttt{e}$ can be composed with $\texttt{Lit}_\texttt{c}$ to create class `Lit` in (1) as follows:

$$\texttt{Lit} = \texttt{Lit}_e \bullet \texttt{Lit}_p \bullet \texttt{Lit}_c$$

For notational simplicity we use, throughout the chapter, symbol • to denote function composition and omit function application as our values are functions without arguments, like $\texttt{Lit}_c$ in this case. We refer to expressions that use operator • as *composition expressions*.

Function composition works well with the order restrictions we identified before. For instance, a composition expression like $\texttt{Lit}_e \bullet \texttt{Lit}_c \bullet \texttt{Lit}_p$ makes no sense because $\texttt{Lit}_p$ is a function not a value and $\texttt{Lit}_p$ references a field defined in $\texttt{Lit}_c$. Similarly, $\texttt{Lit}_e \bullet \texttt{Lit}_p \bullet \texttt{Lit}_c$ is functionality-wise equivalent to $\texttt{Lit}_p \bullet \texttt{Lit}_e \bullet \texttt{Lit}_c$ as we noted in (2) and (3).

Even in this simple example we can identify features (increments in program functionality) to build a product line. The first one is the core functionality (a field and a constructor) of the product line as expressed in $\texttt{Lit}_c$. A second feature is an operation to `print` the field as implemented in $\texttt{Lit}_p$. And the third feature $\texttt{Lit}_e$ is another operation that returns the `value` stored in the class. Using these three features we can create the following programs to build a product line:

$$\texttt{P1} = \texttt{Lit}_c$$

$$\texttt{P2} = \texttt{Lit}_p \bullet \texttt{Lit}_c$$

$$\texttt{P3} = \texttt{Lit}_e \bullet \texttt{Lit}_c$$

$$\texttt{P4} = \texttt{Lit}_e \bullet \texttt{Lit}_p \bullet \texttt{Lit}_c \qquad (7)$$

When viewed in this way, a product line is a set of programs created from all valid composition expressions. A composition expression is valid if it does not violate any constraints of the features it composes. A constraint is a problem domain requirement of features in a product line design. For example, let `f` and `g` be features. Typical constraints relate to composition order (feature `f` must be composed before feature `g` means ...• g •...• f •...), feature inclusion (if `f` is included `g` must be also be included), feature exclusion (if `f` is included `g` must be excluded), feature optionality (if `f` is included `g` may or many not be included), or features that are mandatory (`f` must be included always). For instance,

expression $\texttt{Lit}_e \bullet \texttt{Lit}_c \bullet \texttt{Lit}_p$ is not valid because $\texttt{Lit}_p$ and $\texttt{Lit}_c$ violate composition order constraints: $\texttt{Lit}_p$ refers to `value` defined in $\texttt{Lit}_c$ thus it must appear to the left of $\texttt{Lit}_c$, and $\texttt{Lit}_c$ is a value so it must be the rightmost term in our expression. We elaborate more on this point later in the chapter.

It is the most common case that the implementation of a feature involves more than one class. For example, consider a product line that besides defining class `Lit` and its two operations it also requires the definition of a class that creates objects of `Lit` and tests that their operations work correctly[5]. We call this product line `LitTest` throughout the chapter. We can define such test class and its extensions as follows:

```
class Test {                    // Testc
  Lit t;
  Test() { t = new Lit(3); }
}                                                          (8)
refines class Test {            // Testp
  void testPrint() { t.print(); }
}                                                          (9)
refines class Test {            // Teste
  void testEval()  { System.out.println(t.eval()); }
}                                                         (10)
```

Similar to what we did with class `Lit`, we can create the following test programs using class `Test` and its extensions:

$$\texttt{T1} = \texttt{Test}_c$$

$$\texttt{T2} = \texttt{Test}_p \bullet \texttt{Test}_c$$

$$\texttt{T3} = \texttt{Test}_e \bullet \texttt{Test}_c$$

$$\texttt{T4} = \texttt{Test}_e \bullet \texttt{Test}_p \bullet \texttt{Test}_c \qquad (11)$$

Now the question is: How are `Lit` programs $\texttt{P}_i$ and testing programs $\texttt{T}_j$ related? To answer this question notice that if a program $\texttt{P}_i$ includes a `Lit` feature like `print`

5. This is a common practice in software testing with frameworks such as JUnit.

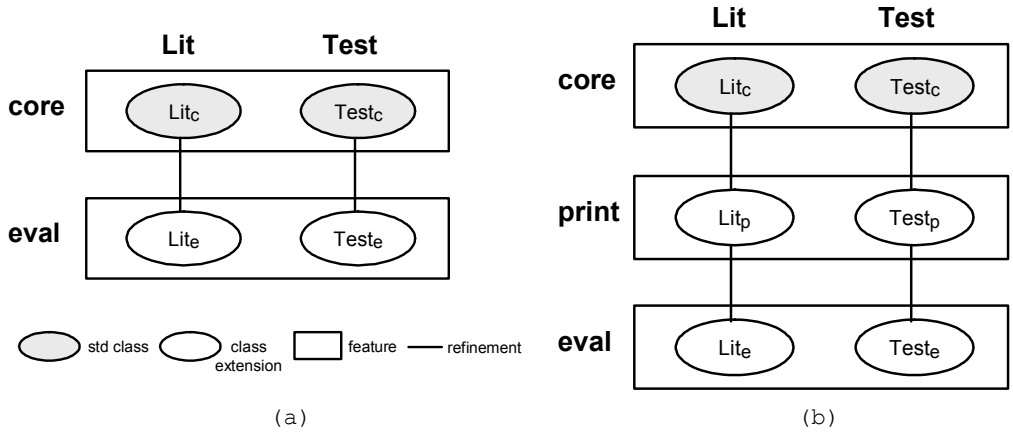


Figure 3.1 Examples of `Lit` and `Test` product line

operation, the related T_j program must have the feature that tests it. Symmetrically, if a program T_j tests for a feature like `print` operation, the program P_i must implement it. In other words, `Lit` and `Test` features are applied lock-step. Thus we can view the `LitTest` product line as having three feature modules: 1) `core` feature with values Lit_c and $Test_c$, 2) `print` feature with functions or class refinements Lit_p and $Test_p$, and 3) `eval` feature with functions Lit_e and $Test_e$. Figure 3.1a depicts the program with features `core` and `eval`, and Figure 3.1b depicts the program with features `core`, `print`, and `eval`. The standard classes or values are slightly shaded and the lines between the features denote the class refinements.

3.2 AHEAD

AHEAD or *Algebraic Hierarchical Equations for Application Design* generalizes GenVoca by considering features as hierarchical modules that can include other artifacts besides source code. In this section we present an algebraic representation of both generalizations and their implementation in the *AHEAD Tool Suite (ATS)*.

3.2.1 Features as Hierarchical Modules

AHEAD redefines GenVoca's notions of *values* as a hierarchical modules (modules that contain other modules) of multiple artifacts, and *functions* to map hierarchical modules to hierarchical modules. In AHEAD, a *model* is a set of features that can either be values or function. For example, a model for `LitTest` in Section 3.1 is denoted as follows:

```
LitTest = { core, print, eval }
```

where the `core` represents classes $\texttt{Lit}_\texttt{c}$ and $\texttt{Test}_\texttt{c}$, print represents $\texttt{Lit}_\texttt{p}$ and $\texttt{Test}_\texttt{p}$, and `eval` represents $\texttt{Lit}_\texttt{e}$ and $\texttt{Test}_\texttt{e}$. This is shown in the following models:

$$\begin{aligned}\texttt{core} &= \{\ \texttt{Lit}_\texttt{c},\ \texttt{Test}_\texttt{c}\ \} \\ \texttt{print} &= \{\ \texttt{Lit}_\texttt{p},\ \texttt{Test}_\texttt{p}\ \} \\ \texttt{eval} &= \{\ \texttt{Lit}_\texttt{e},\ \texttt{Test}_\texttt{e}\ \}\end{aligned} \tag{12}$$

This example illustrates the hierarchical nature of feature modules, where a feature like `core` can itself consist of another set of features such as $\texttt{Lit}_\texttt{c}$ and $\texttt{Test}_\texttt{c}$. Let us now analyzed how feature modules are implemented and composed.

Feature module implementation. We have seen that feature modules are hierarchical and thus can have nested modules. Hence a natural way to implement feature modules is using file directories, where the names of the directories correspond to the names of the features and the leaves of the hierarchy correspond to the source code files as illustrated in Figure 3.2. Please note that: a) we omit feature subscripts in the file names, i.e. $\texttt{Lit}_\texttt{c}$ becomes `Lit`, and b) we include file extension names `.jak` (AHEAD features are implemented in a language called *Jak*, which we describe in Section 3.3). In Section 3.3.3, we describe the roles these two parts play in feature composition.

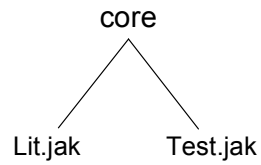


Figure 3.2 Feature `core`

Feature module composition. Consider for instance the program with features `core` and `eval` as illustrated in Figure 3.1a. Let us call this program `CE`. Its composition is thus denoted as:

```
CE = eval • core
```

Given the hierarchical nature of feature modules, the composition also permeates to all their nested features, in other words:

$$CE = eval \bullet core$$
$$= \{ Lit_e, Test_e \} \bullet \{ Lit_c, Test_c \}$$
$$= \{ Lit_e \bullet Lit_c , Test_e \bullet Test_c \}$$

The crucial point to notice is that the composition of nested feature modules is performed based on their names, that is, two nested features are composed if they have the same name. In this example Lit_c with Lit_e and $Test_c$ with $Test_e$. Composition is generalized in the following law.

Law of Composition. Let X and Y be feature modules defined as:

$$X = \{ a_x , b_x , c_x \}$$
$$Y = \{ a_y , c_y , d_y \}$$

where a, b, c, and d are nested features whose subscripts indicate the feature module they belong to. The composition of X and Y, denoted as X•Y, is:

$$X \bullet Y = \{ a_x , b_x , c_x \} \bullet \{ a_y , c_y , d_y \}$$
$$= \{ a_x \bullet a_y , b_x , c_x \bullet c_y , d_y \}$$

Nested features are composed by names (ignoring the subscripts). The features whose names do not have a match, like b_x or d_y, are simply copied.

3.2.2 Features Modules with Non-Code Artifacts

Programs have usually many representations besides source code. Typical representations are UML diagrams, makefiles, XML-based files, grammars, requirements documentation, etc. Consequently, features should also modularize all these types of artifacts.

For example, our LitTest product line may have UML class diagrams associated with classes Lit and Test. This type of diagrams are commonly represented in *XML Metadata Interchange (XMI)*, a standard for representing and manipulating XML data and objects [125]. Thus, feature core with classes Lit and Test and their associated XMI artifacts could be denoted as:

core = { $\texttt{Lit}_c$.jak, $\texttt{Lit}_c$.xmi, $\texttt{Test}_c$.jak, $\texttt{Test}_c$.xmi }

The implementation of this feature is illustrated in Figure 3.3. Note that artifact types are distinguished by the extension names of the files, i.e. XMI artifacts have extension `.xmi`. The Law of Composition establishes that features are composed by their names, therefore the subscripts of the algebraic terms of the features do not form part of the file name so that they could be composed with other features.

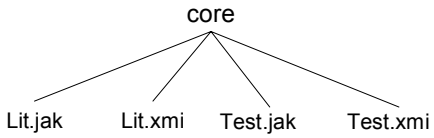


Figure 3.3 Feature `core`

A key element of AHEAD's feature module composition is to treat source code artifacts and non-code artifacts uniformly as stated by the following principle that we explain and illustrate on next section.

Principle of Uniformity. Impose a structure on artifacts that resembles the value and function (refinement) notions of object-based source code.

3.3 AHEAD Tool Suite

The AHEAD Tool Suite (ATS) is an implementation of AHEAD's model of FOP. As mentioned before, the feature modules of ATS tools are implemented using language Jak, an extension of Java [1]. ATS features also contain several types of non-code artifacts. In this section, we describe Jak and illustrate AHEAD's composition for both code and non-code artifacts.

3.3.1 Jak Language

Jak extends Java in several ways [1]. The extensions that concern us are:

Refinement. We have encountered `refines` keyword that indicates a class refinement like $\texttt{Lit}_p$ (5) or $\texttt{Lit}_e$ (6).

Feature containment. Keyword `layer` defines the feature or layer to which a class or a refinement belongs to. For example, the complete code of $\texttt{Lit}_e$ is:

```
layer eval;
refines class Test {
  void testEval()  { System.out.println(t.eval()); }
}                                                          (13)
```

Super construct. The construct `Super` serves two purposes. The first is to indicate reference to superclasses and is equivalent to `super` in Java, but `super` is not a keyword of Jak. The second purpose is to define *method extensions* whereby new functionality is added to a method by means of class refinements[6]. Recall class `Test` in (8) and its refinements. We can, for instance, define a new method `run` to execute tests (method calls) of each refinement as illustrated in Figure 3.4 (we omit `layer` definitions for simplicity). The syntax of `Super` is:

```
Super(paramtypes).method(actparams);
```

where `paramtypes` are the formal parameter types, `method` is the method name and `actparams` are the actual parameters.

The following example illustrates `Super` in method extensions. Consider the execution of method `run` when called on an object of class `Test` defined as

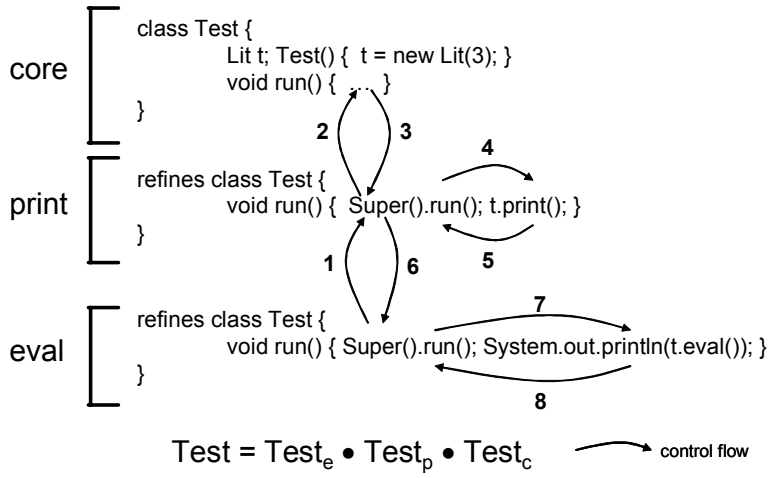


Figure 3.4 `Super` construct example

6. ATS recent release adds `super` for standard inheritance thus leaving `Super` exclusively for method extensions.

$\texttt{Test}=\texttt{Test}_e \bullet \texttt{Test}_p \bullet \texttt{Test}_c$. The most refined method `run` is executed first, in this case the one on feature `eval`. This method in turn calls (with `Super`) the next method extension in the refinement chain (arrow 1), method `run` in `print` which in turns calls method `run` in `core` (arrow 2). After this method is executed, control returns to method `run` in feature `print` (arrow 3) that executes `t.print()` (arrow 4). After the execution of this method (arrow 5), control returns to method `run` in `eval` (arrow 6). Here a `println` statement that displays expression `t.eval()` is executed (arrow 7) and control flow returns to the original call of method `run` (arrow 8). Thus, its execution is similar to `super` in inheritance chains but in this case is according to class refinements.

Another way to visualize composition of method extensions is by aggregating (macro expansion or substitution) to the original method body the bodies of the method extensions according to the order of the composition expression. In our previous example, the extended method `run` is:

```
void run() { … t.print(); System.out.println(t.eval()); }
```

Constructor refinement. It is the counterpart of method extensions. For example, assume class refinement $\texttt{Test}_p$ requires defining another `Test` variable and assigning an object to it. This is expressed as:

```
 refines class Test {          // print feature
   Test t2;
   refines Test() { t2 = new Test(); }
}                                                              (14)
```

Notice that `refines` is used as a modifier to a standard constructor declaration. However, there is no equivalent to referencing higher (in the constructor refinement chain) constructors like using `super` in constructors of Java subclasses. Similarly, there is no equivalent to `this` in Java.

New modifier. This modifier indicates that a method in a class refinement replaces all of its previous definitions in the refinement chain. For example:

```
class A { void m() { println("Hello"); }
refines class A { new void m() { println("Bye");  }
```

When composed, the resulting definition of method `m` is the second one which displays a bye message. In Chapter 6 we present another use of this modifier.

Overrides modifier. This modifier tells programmers that a method of a class refinement is overriding the definition in its superclass. In Chapter 6 we provide examples of its use and its relevance for the implementation of AHEAD product lines using aspects.

3.3.2 Non-Code Artifacts

The Principle of Uniformity establishes that all artifacts must be structured in such a way that they resemble values and functions as described for Object Oriented language source code. This principle implies:

- Treatment of standard non-code artifacts as values.
- Extensions to artifact definitions to accommodate for functions (refinements) and the counterpart of `Super`.
- Implementation of artifact-specific composition tools. This tool provides the polymorphic implementation of operation •. ATS provides an application called `composer` that selects the tool associated with an artifact type and performs the corresponding composition. In the case of Jak code ATS provides two different composition tools.

Let us illustrate these implications with an example of XMI artifacts for class `Lit` in feature `core` and its refinement in feature `eval`. Figure 3.5a shows an overly simplified[7] XMI representation of $\texttt{Lit}_c$ class diagram. Since term $\texttt{Lit}_c$ corresponds to a value, its associated XMI representation must also correspond to a value. Values adhere to the standard specification of their corresponding artifact. For instance, AHEAD source code values are programs that exclusively use Java program constructs available in Jak. Similarly, in case of XMI, values are files that adhere to the XMI specification [125].

7. XMI is a non-trivial and somewhat verbose standard. Further details on the full-fledge representation of these fragments can be derived from the XMI specification [125].

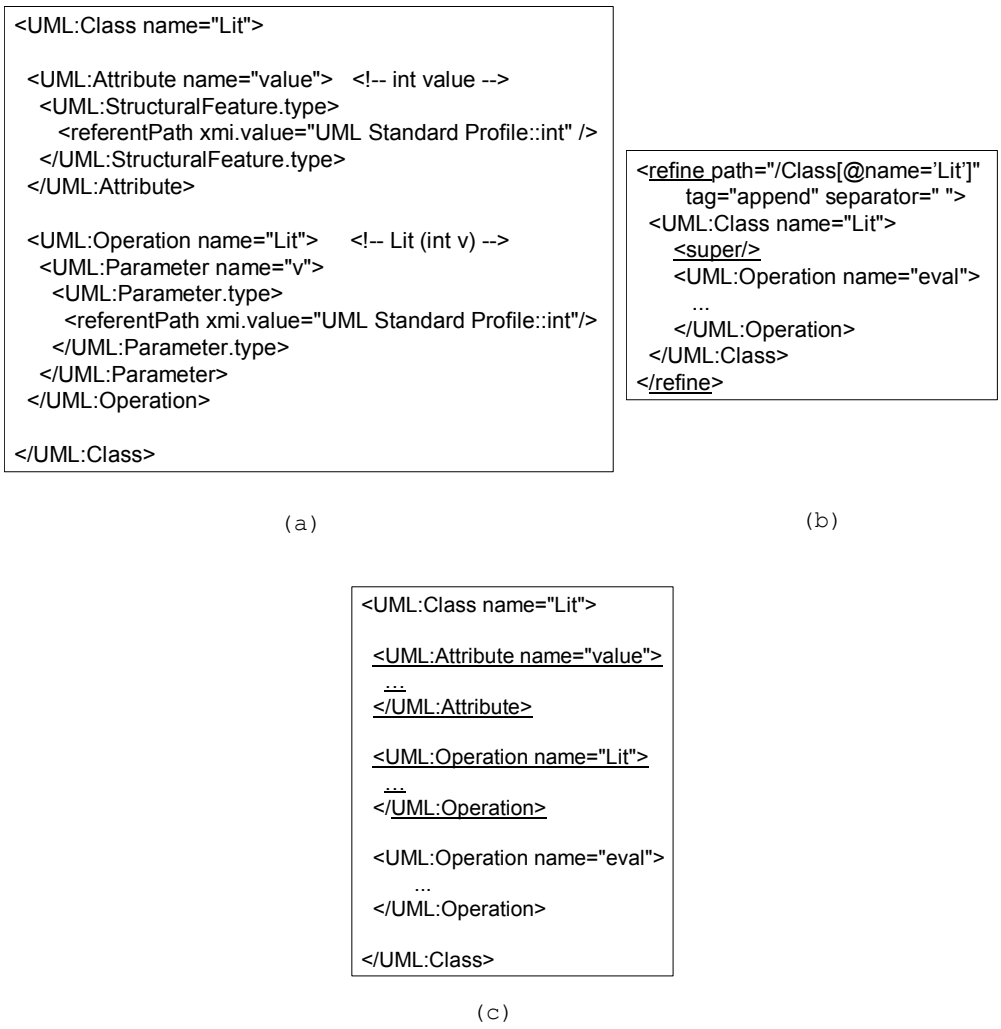


Figure 3.5 Composition of XMI artifacts

We have seen the extensions that Jak makes to Java to denote class refinement and method extension via `Super`. The question is: what are their counterparts in XMI?.

XML composition is implemented in ATS with a tool called *XML Composer (XC)* [1]. XC extends XML files with tags `refine` and `super` as shown underlined in Figure 3.5b for refinement $\texttt{Lit}_e$[8].

8. We elide details of method `eval` as they are very similar to the constructor definition.

Tag `refine` indicates an XML file refinement. It contains three attributes:

- Attribute `path` is an XPath expression that points to the fragment that is refined.
- Attribute `tag` defines the attribute composition policy when there are common attributes in the value or functions files. The possible values are `append`, `prepend` or `override`. Default option is `append`.
- Attribute `separator` specifies a separation string when attributes are appended or prepended. Default is an empty string.

Tag `super` indicates the place where the refined code is going to be inserted. In our example if we compose $\texttt{Lit}_e \bullet \texttt{Lit}_c$, the result obtained is sketched in Figure 3.5c. Note that the underlined code substitutes tag `</super>` in Figure 3.5b.

Currently ATS supports composition of equation files, XML files, and grammar files [1].

3.3.3 Feature Module Composition

Features with non-code artifacts are also composed according to the Law of Composition (Section 3.2.1) with three generalizations:

- Feature terms also include an extension name to specify the type of artifact.
- Composition is performed according to names and artifact types.
- Composition operator • is polymorphic on the artifact type, in other words, each artifact implements composition differently.

For example, consider composition `eval` • `core` as defined in (12) with Jak and XMI class diagrams:

$$
\begin{aligned}
\texttt{eval} \bullet \texttt{core} &= \{\ \texttt{Lit}_e\texttt{.jak},\ \texttt{Lit}_e\texttt{.xmi},\ \texttt{Test}_e\texttt{.jak},\ \texttt{Test}_e\texttt{.xmi}\ \} \bullet \\
&\quad \{\ \texttt{Lit}_c\texttt{.jak},\ \texttt{Lit}_c\texttt{.xmi},\ \texttt{Test}_c\texttt{.jak},\ \texttt{Test}_c\texttt{.xmi}\ \} \\
&= \{\ \texttt{Lit}_e\texttt{.jak} \bullet \texttt{Lit}_c\texttt{.jak},\ \texttt{Lit}_e\texttt{.xmi} \bullet \texttt{Lit}_c\texttt{.xmi}, \\
&\quad\ \texttt{Test}_e\texttt{.jak} \bullet \texttt{Test}_c\texttt{.jak},\ \texttt{Test}_e\texttt{.xmi} \bullet \texttt{Test}_c\texttt{.xmi}\ \}
\end{aligned}
$$

In Section 3.3.1 we illustrated composition of Jak files and in Figure 3.5 we described the composition of $\texttt{Lit}_e\texttt{.xmi} \bullet \texttt{Lit}_c\texttt{.xmi}$. The rest of the terms are composed similarly.

3.4 Origami

So far we have seen composition examples that consist of a few features with simple dependencies. As expected, interesting feature-based applications are constructed with larger number of feature modules with complex relations among them. Specifying program synthesis as composition of feature modules quickly becomes unwieldy because the number of feature combinations explodes as the number of features increases. This problem requires an architectural model to simplify program specification and thus enable AHEAD to synthesize product lines of non-trivial scale and complexity. *Origami* is an architectural model of AHEAD that tackles this problem.

In the following sections we motivate and illustrate the need of Origami; we explain how it works, describe its algebraic representation, and show its connections with other research.

3.4.1 Motivation

Our example product line `LitTest` contains only three features, copied from (12):

$\texttt{core} = \{ \texttt{Lit}_c, \texttt{Test}_c \}$

$\texttt{print} = \{ \texttt{Lit}_p, \texttt{Test}_p \}$

$\texttt{eval} = \{ \texttt{Lit}_e, \texttt{Test}_e \}$

Each feature contains two classes or class refinements. If we define `Test` with method run and its extensions as in Figure 3.4, we can synthesize five different programs. We can obtain their composition expressions rather easily as there is only one value (feature `core`), and the two function features can be permuted.

Unfortunately specification simplicity degrades rapidly even for product lines slightly larger. Consider what happens if we break `LitTest` features into their constituent

classes and class refinements. We call this new product line LitTest′ and define its features as follows:

$$\texttt{lc} = \{\ \texttt{Lit}_c\ \},\ \texttt{tc} = \{\ \texttt{Test}_c\ \}$$

$$\texttt{lp} = \{\ \texttt{Lit}_p\ \},\ \texttt{tp} = \{\ \texttt{Test}_p\ \}$$

$$\texttt{le} = \{\ \texttt{Lit}_e\ \},\ \texttt{te} = \{\ \texttt{Test}_e\ \} \qquad (15)$$

As a naming convention, we form the names of these feature modules with the first letter of the class followed by the first letter of the operation they implement. For instance, feature module lp implements in class Lit operation print. Similarly, feature module te tests in class Test operation eval.

Additionally, we put a further restriction: synthesize test features optionally in accordance with the selection of Lit operations. For example, if a program implements operations print and core for Lit and the synthesis of tests is selected, features tp and tc must be generated.

The natural questions to ask are: What are all the valid composition expressions of LitTest′?, and how many different programs can be generated? After some careful, laborious and error-prone process it is possible to derive all twenty two valid compositions of LitTest′ illustrated in Figure 3.6.

The table shows for each selection of Lit features, the valid composition expressions with and without tests. For instance, if core is selected for Lit (feature module lc) with test option (feature module tc) the program generated is expressed as tc • lc. Besides illustrating all valid program compositions, the table also shows the different programs that can be generated. Table entries with more than one expression indicate that a program can be generated via different compositions[9]. For example, the program with features core (lc) and print (lp) with tests (tc, tp) can be generated in two different ways: tp • tc • lp • lc or tp • lp • tc • lc. In other words, both composition expressions

9. In Figure 3.6, composition expressions lp • le • lc and lp • le • lc are in the same table entry because they generate the same program. These expressions are aligned with their associated test-augmented expressions which in this case generate two different programs (different execution order of method run).

Lit Features	Test Features	
	No Test	Test
core	lc	tc • lc
core, print	lp • lc	tp • tc • lp • lc , tp • lp • tc • lc
core, eval	le • lc	te • tc • le • lc , te • le • tc • lc
core, print, eval	le • lp • lc	te • le • tp • lp • tc • lc , te • le • tp • tc • lp • lc, te • tp • le • tc • lp • lc , te • tp • tc • le • lp • lc
	lp • le • lc	tp • te • le • tc • lp • lc , tp • te • tc • le • lp • lc
		tc • lp • te • le • tc • lc , tp • lp • te • tc • le • lc, tp • te • lp • tc • le • lc , tp • te • tc • lp • le • lc
		te • tp • lp • tc • le • lc , te • tp • tc • lp • le • lc

Figure 3.6 Program compositions in `LitTest'`

produce class `Lit` with `value` field, constructor and `print` method; and class `Test` with field `t` of type `Lit`, a constructor, method `run` which is extended with a call to method `print` on `t`.

Even our simple product line `LitTest'`, formed with only six features, evidences the complexity of program specification. The problem is summarized in the question: how to devise concise and valid composition expressions from a selection of features in a systematic and principled way?

Next section explains how Origami addresses this question.

3.4.2 Origami Matrices

An *Origami Matrix* is a n-dimensional matrix where each dimension is formed by a set of features and its elements are the feature modules that implement the functionality at the intersection of their coordinates in the n-dimensions.

We can represent `LitTest'` with the two-dimensional origami matrix in Figure 3.7. The horizontal dimension (rows) corresponds to operations `core`, `print`, and `eval`. The vertical dimension (columns) corresponds to data types classes `Lit` and `Test`. In origami matrices each entry implements the functionality at the intersection of its feature coordinates. For instance, feature module `le` implements `eval` (coordinate in operations dimension) in class `Lit` (coordinate in classes dimension).

	Lit	Test
core	lc	tc
print	lp	tp
eval	le	te

Figure 3.7 `LitTest` Matrix

We define an AHEAD model for each dimension:

```
Operation = { core, print, eval }
DataType = { Lit, Test }                                    (16)
```

Composition in an Origami matrix is performed by composing column-wise or row-wise, a process called *folding*, the selected features according to the composition constraints of the matrix. For `Lit-Test'` the constraints are depicted in Figure 3.8. The full arrows indicate refinement dependencies (a feature refines another feature) and dashed arrows indicate reference dependencies (a feature references entities declared at another feature). For instance, feature module `tp`: a) refines class `Test` thus it needs module `tc`, b) calls `print` method in an object of class `Lit` thus it refers that method of module `lp`. The rest of the product line constraints have similar interpretations.

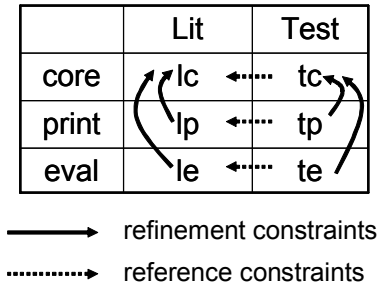


Figure 3.8 Composition Constraints

Besides refinement constraints and reference constraints there exist design constraints that capture semantic relationships among features. AHEAD implements design constraints with an artifact called *design rules* (extension `.drc`), a domain-specific language. For further details consult [1].

Let us illustrate how Origami composition works. For example, we want to synthesize the program that implements `eval` and also performs tests. In our matrix this

means selecting rows `core` and `eval`, and columns `Lit` and `Test`. Because we are only interested in these feature subsets of both dimensions, we can project the selected rows and columns and eliminate those not required. This is illustrated for our example in Figure 3.9a where row `print` was eliminated. At this point there are two alternatives. The first alternative folds rows first, see Figure 3.9b. This folding satisfies the composition constraints, `le` is composed after `lc` and `te` is composed after `tc`, thus it is a valid folding. The second step folds the matrix by columns, see Figure 3.9c. This folding is valid because satisfies composition constraints, `tc` is composed after `lc` and `te` is composed after `le`. Since the resulting matrix has a single entry, it cannot be folded any further and the resulting expression denotes a valid composition expression for the selected features. The second alternative for our example folds columns first, Figure 3.9d. This folding is valid because it satisfies composition constraints, `tc` is composed after `lc` and `te` is composed after `le`. The second step folds rows. This folding is valid because it satisfies that `le` is composed after `lc` and `te` is composed after `tc`. This alternative yields a different valid composition expression.

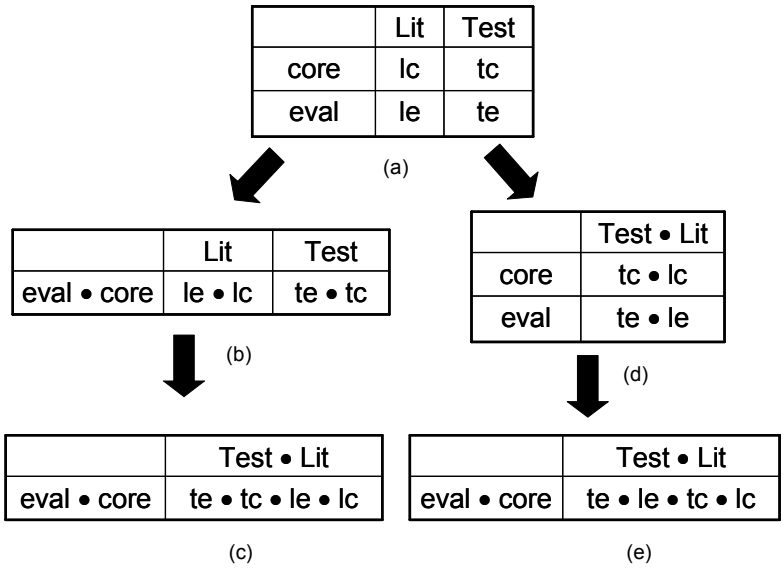


Figure 3.9 Example of Origami Composition

In summary, for our application example two valid composition expressions can be derived:

```
te • tc • le • lc  and  te • le • tc • lc                    (17)
```

The question now becomes: what do we gain with Origami? Origami provides a disciplined and systematic way to derive valid composition expressions given a set of selected features and composition constraints. But most importantly, Origami simplifies program specification by describing programs with *dimensional equations*, composition expressions of features from dimensional models, instead of the feature modules. For example, a program defined in (17) can be specified with two dimensional equations that compose features of dimensional models in (16):

```
operation = eval • core
datatype = Test • Lit                                        (18)
```

Where `operation` is the equation for the selected features in the operation dimension and `class` is the equation for the classes dimension. Certainly for our small example the simplification is not significant but consider the general case where the matrix has `k` dimensions and `n` features per dimension. For that case, the specification of a program using its feature modules is $O(n^k)$, whereas using one equation per dimension the specification is $O(nk)$.

Origami matrices are related to the extensibility problem, also known as the "expression problem" [48][155], a fundamental problem of software design that consists of extending a data abstraction to support a mix of new operations and data representations. On Chapter 4 we explore this relationship further.

3.4.3 Origami as a Multi-Dimensional Algebraic Model

A fundamental principle of Software Engineering is *Separations of Concerns (SoC)* which proposes breaking program development into units of interest or functionality (concerns) as a mechanism to tackle program complexity [54]. *Multi-Dimensional Separation of Concerns (MDSoC)* generalizes this principle to advocate that modularity can be understood by multi-dimensional spaces of units (concerns), where dimensions represent differ-

ent modularizations and units along the dimensions are particular instances of that dimension's modularity [150][28]. Origami models are examples of the MDSoC where each dimension is formed by an AHEAD model (a set of dimension features).

Origami matrices also have an algebraic representation. Consider again program formed by class `Lit` with method `eval` and corresponding test. We presented in (16) the AHEAD models of our product line and in (18) the dimensional equations that specify this program. We represent each dimensional equation as a summation of features from each corresponding AHEAD model:

$$\texttt{Operation} = \{\texttt{core, print, eval}\}\ \texttt{DataType} = \{\texttt{Lit, Test}\}$$

$$\texttt{operation} = \texttt{eval} \bullet \texttt{core} = \Sigma_{i\varepsilon(\texttt{eval,core})}\texttt{Operation}$$

$$\texttt{datatype} = \texttt{Test} \bullet \texttt{Lit} = \Sigma_{i\varepsilon(\texttt{Lit,Test})}\texttt{DataType}$$

A program is represented as summations, one for each dimension, ranging over the selected features of the product line matrix (`MLitTest'` in our example). Thus our program is represented as:

$$\texttt{P} = \Sigma_{i\varepsilon(\texttt{eval,core})}\Sigma_{i\varepsilon(\texttt{Lit,Test})}\texttt{MLitTest'}_{\texttt{operation,datatype}}$$

The algebraic representation of origami matrices has proven an useful abstraction to analyze matrix orthogonality, a property that guarantees that the same program is produced for any summation order [22].

3.4.4 Scaling Origami

Origami originated during the development of *Jedi*, a documentation generator for extensible domain-specific languages [103]. The goal of this tool was to replicate the functionality of generating HTML documents provided by Sun Microsystem's `javadoc` [84] but for extensions of Java required by AHEAD. As the development proceeded it became clear that an underlying architecture was emerging for the implementation of each language extension. Each extension consists of three parts:

- *Parse*: codes that parses the language constructs made by an extension and creates a parse tree.

- *Harvest*: reads a program with its comments and creates a data structure or repository with them.
- *Doclet*: or documentation generation, produces HTML code for a program based on the information harvested and document templates.

Figure 3.10 illustrates the architecture of Jedi. As we mentioned before, AHEAD makes several extensions to Java (`Java`). Two of those extensions are support for state machines (`Sm`) and feature module (`Tmpl`) declarations – `layer` constructs originally called templates. Considering these two extensions we have the origami matrix shown in Figure 3.11. In this matrix, the name of the feature module is formed with the first letter of the extension and an architecture part. Like in the matrix of `LitTest`, the Jedi matrix is composed with row and column foldings that satisfy the set of composition constraints of the matrix. For example, a valid expression for the three extensions (`Java`, `Tmpl`, `Sm`) and the three Jedi architecture parts (`Parse`, `Harvest`, `Doclet`) is:

	Parse	Harvest	Doclet
Java	JParse	JHarvest	JDoclet
Tmpl	TParse	THarvest	TDoclet
Sm	SParse	SHarvest	SDoclet

Figure 3.11 Jedi Origami Matrix

```
Jedi =   SDoclet • SHarvest • SParse •
           TDoclet • THarvest • TParse •
             JDdoclet • JHarvest • JParse                (19)
```

This program results from: a) folding column `Harvest` on `Parse`, b) folding column `Doclet`, c) folding row `Tmpl` on `Java`, d) folding row `Sm`. The same program can be expressed with a shorter specification using its dimensional equations:

```
Language = Sm • Tmpl • Java
Architecture = Doclet • Harvest • Parse                  (20)
```

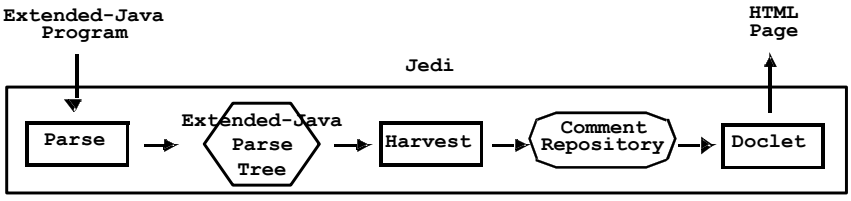


Figure 3.10 Jedi Architecture

Jedi is a testament to the scalability of Origami matrices because it is implemented with 30K+ LOC, in several hundred classes, interfaces, and refinements. An even larger example is AHEAD tool suite. It is generated from a 3-dimensional Origami matrix with 14 language extension features, 9 architecture features, and 9 language interaction features. The total code generated for ATS is 250K+ LOC. For further details on the construction of ATS consult [1][30].

3.5 Summary

In this chapter we presented the foundations of Feature Oriented Programming. We described GenVoca, the precursor of FOP, and how it was extended by AHEAD to include hierarchical feature modules that can contain multiple artifacts. We showed how these two extensions are algebraically represented and implemented, guided by the Law of Composition and the Principle of Uniformity.

We exemplified how program specification becomes harder as the number of feature modules increases. We illustrated how Origami addresses this problem, its composition mechanism and underlying algebraic representation, and how it scales program synthesis from simple product lines to tools and tool suites like ATS.

Chapter 4

Modularization Technologies Evaluation

In Chapter 2 we saw the importance of features in product line engineering. In Chapter 3 we presented Feature Oriented Programming and AHEAD as an approach to modularize and compose features. In recent years, other approaches have been proposed that have the potential to provide better support for feature modularity. These technologies have very different notions of modularity and composition, and as a consequence are difficult to compare and unify. Thus it is increasingly important to advance standard problems and metrics for technology evaluation. A few attempts have been made to compare technologies and evaluate their use to refactor and re-implement systems that are not part of a product family [41][58][112][123]. But for a few studies [49][161][120], the use of new technologies to modularize features in a product line is largely unexplored.

In this chapter we present a standard problem that exposes common and fundamental issues that are encountered in feature modularity in product-lines[1]. The problem reveals technology-independent properties that feature modules should exhibit. We use these properties to evaluate solutions written in six modularization technologies: AspectJ [11][87], Hyper/J [130][151], Jiazzi [113][114][161], Scala [143][126][127][128], CaesarJ [8][119][120], and AHEAD [1][30]. Chapter 5 goes a step further by giving the properties of this evaluation an algebraic foundation. Such algebraic model of feature modules and their composition can provide a framework or set of criteria that a rigorous mathematical

1. The core of this chapter comes from our ECOOP paper [106].

presentation should satisfy and can help reorient the focus on clean and mathematically justifiable abstractions when developing new tool-specific concepts.

4.1 The Expressions Product Line

The *Expressions Product Line (EPL)* is based on the extensibility problem also known as the "expression problem" [48][155]. It is a fundamental problem of software design that consists of extending a data abstraction to support a mix of new operations and data representations. It has been widely studied within the context of programming language design, where the focus is achieving data type and operation extensibility in a type-safe manner. Rather than concentrating on that issue, we consider the *design and synthesis* aspects of the problem to produce a family of program variations. More concretely, what features are present in the problem? How can they be modularized? And how can they be composed to build all the programs of the product-line?

4.1.1 Problem Description

Our product-line is based on Torgersen's expression problem [153]. Our goal is to define data types to represent expressions of the following language:

```
Exp   :: = Lit | Add | Neg
Lit   :: = <non-negative integers>
Add   :: = Exp "+" Exp
Neg   :: = "-" Exp
```

Two operations can be performed on expressions of this grammar:

1) `Print` displays the string value of an expression. The expression `2+3` is represented as a three-node tree with an `Add` node as the root and two `Lit` nodes as leaves. The operation `Print`, applied to this tree, displays the string "`2+3`".

2) `Eval` evaluates expressions and returns their numeric value. Applying the operation `Eval` to the tree of expression `2+3` yields `5` as result.

```
lp  interface Exp {
lp    void print();
le    int eval();
lp  }

ap  class Add implements Exp {
ap    Exp left, right;
ap    Add (Exp l, Exp r) {
ap     left = l; right = r; }
ap    void print() {
ap      left.print();
ap      System.out.print("+");
ap      right.print();
ap    }
ae    int eval() {
ae      return left.eval()
ae           + right.eval();
ae    }
ap  }

np  class Neg implements Exp {
np    Exp expr;
np    Neg (Exp e) { expr = e; }
np    void print() {
np      System.out.print("-(");
np      expr.print();
np      System.out.print(")");
np    }
ne    int eval() {
ne      return expr.eval() * -1;
ne    }
np  }
```

```
lp  class Lit implements Exp {
lp    int value;
lp    Lit (int v) { value = v; }
lp    void print() {
lp      System.out.print(value);
lp    }
le    int eval() { return value; }
lp  }

lp  class Test {
lp    Lit ltree;
ap    Add atree;
np    Neg ntree;
lp    Test() {
lp      ltree = new Lit(3);
ap      atree = new Add(ltree, ltree);
np      ntree = new Neg(ltree);
lp    }
lp    void run() {
lp      ltree.print();
ap      atree.print();
np      ntree.print();
le      System.out.println(ltree.eval());
ae      System.out.println(atree.eval());
ne      System.out.println(ntree.eval());
lp    }
lp  }
```

Figure 4.1 Complete code of the Expressions Product Line

We add a class `Test` that creates instances of the data type classes and invokes their operations. We include this class to demonstrate additional properties that are important for feature modules. Figure 4.1 shows the complete Java code for a program of the product-line that implements all the data types and operations of EPL. Shortly we will see what the annotations at the beginning of each line mean.

	Operations		Data types		
Program	Print	Eval	Lit	Add	Neg
1	✓		✓		
2	✓	✓	✓		
3	✓		✓	✓	
4	✓	✓	✓	✓	
5	✓		✓		✓
6	✓	✓	✓		✓
7	✓		✓	✓	✓
8	✓	✓	✓	✓	✓

Figure 4.2 Members of the EPL

From a product line perspective, we can identify two different feature sets [51]. The first is that of the operations {`Print`, `Eval`}, and the second is that of the data types {`Lit`, `Add`, `Neg`}. Using these sets, it is possible to synthesize all members of the product-line described in Figure 4.2 by selecting one or more operations, and one or more data types.

For instance, row 4 is the program that contains `Lit` and `Add` with operations `Print` and `Eval`. As with any product line design, in EPL there are constraints on how features are combined to form programs. For example, all members require `Lit` data type, as literals are the only way to express numbers.

A common way to implement features in software product lines is to use preprocessor declarations to surround the lines of code that are specific to a feature. If we did this for the program in Figure 4.1, the result would be unreadable. Instead, we use an annotation at the start of each line to indicate the feature to which the line belongs. This makes it easy to build a preprocessor that receives as input the names of the desired features and strips off from the code of Figure 4.1 all the lines that belong to unneeded features. As we saw in Chapter 2, this approach is very brittle for problems of larger scale and complexity. Never the less, the approach can be used as a reference to define what is expected from feature modules in terms of functionality (classes, interfaces, fields, methods, constructors), behaviour (sequence of statements executed), and composition.

4.1.2 Feature modularization

A natural representation of the expression problem, and thus for EPL, is a two-dimensional matrix [48][155][65]. The vertical dimension specifies data types and the horizontal dimension specifies operations. Each matrix entry is a *feature module* that implements the operation, described by the column, on the data type, specified by the row. As a naming convention throughout the chapter, we identify matrix entries by using the first letters of the row and the column, e.g., the entry at the intersection of row `Add` and column `Print` is named `ap` and implements operation `Print` on data type `Add`. This matrix is shown in Figure 4.3 where module names are encircled.

To compose any program from Figure 4.2, the modules involved are those at the intersection of the selected columns and the selected rows. For example, program number 1, that provides `Print` operation on `Lit`, only requires module `lp`. Another example is program 6, that implements operations `Print` and `Eval` on `Lit` and `Neg` data types, requires modules `lp`, `le`, `np`, and `ne`.

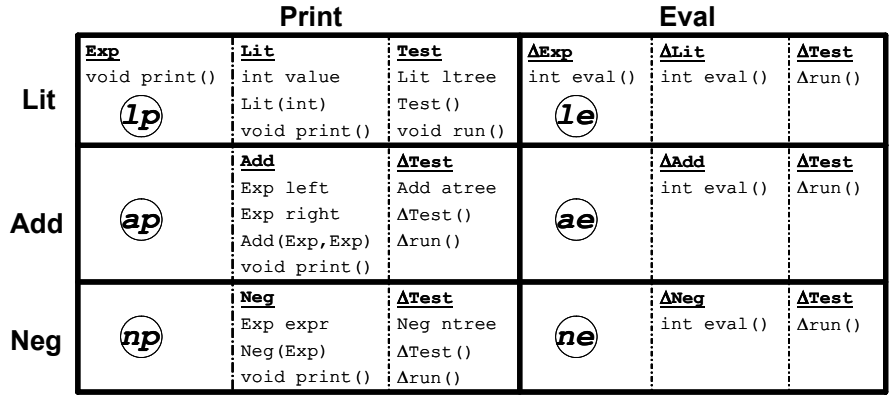


Figure 4.3 Matrix representation and Requirements

The source code of a feature module are the lines that are annotated with the name of the module. For instance, the contents of feature `ap` include:

a) Class `Add` with `Exp` fields `left` and `right`, a constructor with two `Exp` arguments, and method `void print()`, and

b) An increment to class `Test`, because it is adding something to the class as opposed to contributing a brand new class as is the case of class `Add`. This increment is symbolized by `ΔTest` in Figure 4.3. It adds: field `atree`, a statement to the body of the constructor expressed with `ΔTest()`, and a statement to the body of method `run` expressed as `Δrun()`.

For clarity the `Exp` interface was put inside module `lp` instead of creating a separate row for it. This decision makes sense since the other data types are built using `Lit` objects. Also, we put the constructors and fields of the data types in column `Print` instead of refactoring them into a new column and have columns `Print` and `Eval` implement only their corresponding methods. Additionally, from the design requirements we can infer dependencies and interactions among the feature modules. For instance, if we want to build a program with module `ap`, we also need to include module `lp` because `ap` increments the `Test` class which is introduced in `lp`. Later, we briefly discuss this issue as compositional constraints, which are not the focus of this chapter. Constraints are discussed in [1][25][35].

4.2 Basic Properties for Feature Modularity

To give structure to our evaluation, we identify a set of basic properties about features that can readily be inferred from, illustrated by, and assessed in EPL and its solutions in the five technologies evaluated. Conceivably, there are other desirable properties that feature modules should exhibit such as readability, ease of use, etc. However, for sake of simplicity and breadth of scope, they are not part of this evaluation as their objective assessment would require a larger case study that would prevent us from comparing all five technologies together.

The properties are grouped into two categories, covering the basic definition of features and their composition to create programs. The first properties in each category follow from the structure of EPL, while the others come from the studied solutions to EPL and are desirable from the software engineering perspective.

4.2.1 Feature Definition Properties

The first category of properties relate to the definition of the basic building blocks of EPL, the representation of each piece, and their organization into features.

Program deltas. The code in Figure 4.1 can be decomposed into a collection of *program deltas* or program fragments. The kinds of program deltas required to solve EPL are shown in Figure 4.3, and include:

- *New Classes*, for example `Lit` in module `lp`.
- *New Interfaces*, for example `Exp` in module `lp`.
- *New fields* that are added to existing classes, like field `atree` in module `ap` is added to class `Test`.
- *New methods* that are added to existing interfaces, like `eval()` in module `le` is added to interface `Exp`.
- *Method extensions* that add statements to methods. For example, extension to method `run()`, expressed by `Δrun()`, in all modules except `lp`.

- *Constructor extensions* that add statements to constructors. For instance, extensions to constructor `Test()`, expressed by `ΔTest()`, in modules `ap` and `np`.

There are other program deltas, such as new constructors, new static initializers, new exception handlers, etc. that are not needed for implementing EPL and thus are not considered in this evaluation. Nonetheless, we believe that EPL contains a sufficient set of program deltas for an effective evaluation.

Cohesion. It must be possible to collect a set of program deltas and assign them a name so that they can be identified and manipulated as a cohesive module.

Separate compilation. Separate compilation of features is useful for two practical reasons: a) it allows debugging of feature implementation (catching syntax errors) in isolation, and b) it permits the distribution of bytecode instead of source code.

4.2.2 Feature Composition Properties

Once a set of feature modules has been defined, it must be possible to compose them to build all the specific programs in the Expression Product Line.

Flexible Composition. The implementation of a feature module should be syntactically independent of the composition in which it is used. In other words, a fixed composition should not be hardwired into a feature module. Flexible composition improves reusability of modules for constructing a family of programs.

Flexible Order. The order in which features are composed can affect the resulting program. For instance, in EPL, the order of test statements in method `run()` affects the output of the program. The program in Figure 4.1 is the result of one possible ordering of features, namely (`lp`, `ap`, `np`, `le`, `ae`, `ne`). Another plausible order in EPL is to have expressions printed and evaluated consecutively, as in order (`lp`, `le`, `ap`, `ae`, `np`, `ne`). Hence, feature modules should be composable in different orders.

Closure under Composition. Feature modules are closed under composition if one or more features can be composed to make a new composite feature. Composite features must be usable in all contexts where basic features are allowed. In EPL, it would be

natural to compose the `Lit` and `Neg` representations to form a `LitNeg` feature which represents positive and negative numbers.

Static Typing. Feature modules and their composition are subject to static typing which helps to ensure that both are well-defined, for example, preventing method-not-found errors. We base the evaluation of this property on the availability of a formal typing theory or mechanism behind each technology.

Using these properties we evaluate AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD in the following sections.[2] We use a concrete example to illustrate these alternatives, i.e. the program that supports `Print` and `Eval` operations in `Lit` and `Add` data types (program number 4 in Figure 4.2). Thus, the program has four modules: `lp`, `ap`, `le`, and `ae` that we compose in this order (the same as in Figure 4.1). Throughout the chapter, we call this program `LitAdd`.

4.3 AspectJ

An aspect, as implemented in *AspectJ*[3] [11][87], modularizes a *crosscut* as it contains code that can extend several classes and interfaces. AspectJ has two types of crosscuts: static and dynamic. *Static crosscuts* affect the static structure of a program [11][93]. Examples are *introductions*, also known as *inter-type declarations*, that add fields, methods, and constructors to existing classes and interfaces.

Dynamic crosscuts run additional code when certain events occur during program execution. The semantics of dynamic crosscuts are commonly understood and defined in terms of an event-based model [94][157]. As a program executes, different events fire. These events are called *join points*. Examples of join points are: variable reference, variable assignment, execution of a method body, method call, etc. A *pointcut* is a predicate that selects a set of join points. *Advice* is code executed before, after, or around each join point matched by a pointcut.

2. For a more detailed description of the implementation see [107].
3. We used AspectJ version 1.1 for our evaluation.

4.3.1 Feature modules and their composition

The implementation of module `lp` is straightforward as it consists of Java interface `Exp` and classes `Lit` and `Test`. In AspectJ literature, programs written using only pure Java code are called *base code*. In Figure 4.4a, the names of files that are base code are shown in italics, while those of *aspect code* are shown in all capital letters.

	Print	Eval
Lit	*Exp, Lit, Test*	LE
Add	*Add*, AP	AE
Neg	*Neg*, NP	NE

(a)

```
public aspect LE {

  // ΔExp interface
  public abstract int Exp.eval();

  // ΔLit class
  public int Lit.eval() { return value; }

  // ΔTest, advice that implements Δrun()
  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());
  }
}
```

(b)

Figure 4.4 AspectJ Solution

Alternatively, we could have declared the new classes and interfaces as nested elements of an aspect. However, they would be subject to the instantiation of their containing aspect, and their references would be qualified with the aspect name where they are declared. For these reasons, we decided to implement classes and interfaces in separate files.

From Figure 4.3, module `le`:

1) adds method `eval()` to interface `Exp`,

2) adds the implementation of `eval()` to class `Lit`, and

3) appends a statement to method `run()` of class `Test` that calls `eval()` on field `ltree`.

The entire code of module `le` is implemented with the aspect shown in Figure 4.4b. The first two requirements use AspectJ's introductions [11][87][4]. Method extensions, like that of the third requirement, cannot be implemented as introductions because there is already a member of that class with the same signature. Hence, to implement the last requirement it is necessary to utilize AspectJ's pointcuts and advice.

Since it is required to execute an additional statement when method `run()` is executed, we must capture the join point of the `execution` of that method. Also, since the statement to add is a method call on field `ltree` of class `Test`, we must get a hold of the object that is the `target` of the execution of method `run()` to access its `ltree` field. These two conditions are expressed in pointcut `LPRun` of Figure 4.4b, where `t` is the reference to the target object. Lastly, to add the extension statement we use an `around` advice. This type of advice executes instead of the join points of the pointcut, but it allows its execution by calling AspectJ's special method `proceed`. We add the new statement to `run()` after the call to method `proceed(t)`.[5]

The implementation of feature module `ap` (not shown in Figure 4.4) uses two files. The first is a Java class to implement data type `Add`. The second is an aspect to implement the extensions to class `Test`. The first extension adds a new field to class `Test`. This is done also using inter-type declaration in the following way:

```
public Add Test.atree;
```

The other two extensions of module `ap`, `ΔTest()` and `Δrun()`, are implemented in a similar way to those of module `le`. The other modules `ae`, `np`, and `ne` have an analogous implementation.

To compose program `LitAdd`, the AspectJ compiler (*weaver*) `ajc`, requires the file names of the base code and the aspects of the feature modules. The composition is specified as follows, where the order of the terms is inconsequential:

---

4. We could also implement the first requirement as follows:

   ```
   public int Exp.eval() { return 0; }
   ```

   This alternative defines a default value for the method which can be subsequently overridden by each class that implements Exp.

5. Method `proceed`, has the same arguments as the advice where it is used.

```
ajc Exp.java Lit.java Test.java LE.java Add.java AP.java
    AE.java  -outjar LitAdd.jar                              (21)
```

The static crosscutting model of AspectJ has a simple realization, after composition there can be only one member for a given signature. However, in the case of dynamic crosscutting, i.e. pointcuts and advice, several pieces of advice can apply to the same join point. In such cases, the order in which advice code is executed is in general undefined[6]. This means that a programmer cannot know a priori, by simply looking at the pointcut and advice code, in what order advice is applied. In program `LitAdd`, this issue is manifested in the order of execution of method `run()` and its extensions. The order that we want is that of Figure 4.1, namely, first the statement from `lp` followed by those of `ap`, `le` and `ae`. However, the order obtained by executing the program is statements from `lp`, `ae`, `ap`, and `le`[7].

AspectJ provides a mechanism to give precedence to advice, thus imposing an order, at the aspect level. In other words, it can give precedence to all the advice of an aspect over those of other aspects. To obtain the order that we want for method `run()`, we must define the following aspect:

```
public aspect Ordering {
   declare precedence : AE, LE, AP;
}
```

And add it to the list of files in the specification (21). For further details on how precedence clauses are built, consult [11][93].

4.3.2 Evaluation

Feature definition. AspectJ can describe all program deltas required for EPL. However, in cases like module `ap` which is implemented with class `Add` and aspect `AP`, there is no way to express that both together form feature `ap`. In other words, AspectJ does

6. There are special rules that apply for certain types of advice when advices are defined in either the same aspect or in others [11]. These rules help determine the order in few cases but not in general.
7. In AspectJ version 1.1

not have a cohesion mechanism to group all program deltas together and manipulate them as a single module. Nonetheless, this issue can be addressed with relatively simple tool support. Aspects cannot be compiled separately, as they need base code in which to be woven.

Feature composition. AspectJ provides flexible composition and order. It can be used to build all members of EPL in the order described in an auxiliary aspect that contains a `declare precedence` clause. This type of clause can also be used inside aspects that implement feature modules, like `LE`, but doing that could reduce order flexibility as the order could be different for different programs where `LE` is used. Feature modules implemented in AspectJ are not closed under composition for two reasons: the absence of a cohesion mechanism and the lack of a general model of aspect composition. The latter is subject of intensive research [57]. Static typing support for AspectJ is also an area of active research [83][156].

4.4 Hyper/J

Hyper/J [151] is the Java implementation of an approach to *Multi-Dimensional Separation of Concerns* (*MDSoC*) called Hyperspaces [130][151]. A *hyperspace* is a set of units. A unit can be either primitive, such as a field, method, and constructor; or compound such as a classe, interface, and package.

A *hyperslice* is a modularization mechanism that groups all the units that implement a *concern* (a feature in this paper) which consists of a set of classes and interfaces. Hyperslices must be *declaratively complete*. They must have a declaration, that can be incomplete (stub) or abstract, for any unit they reference. Hyperslices are integrated in *hypermodules* to build larger hyperslices or even complete systems.

4.4.1 Feature modules and their composition

The Hyper/J weaver performs composition at the bytecode level which makes a natural decision to implement each hyperslice (feature module) as a package that can be compiled independently. Hyperslices that contain only new classes and interfaces, like module `lp`, have a straightforward implementation as Java packages. The interesting case is hyperslices that extend units in other hyperslices. For example, Figure 4.5a shows the package that implements feature `le`. It adds method `eval()` to `Exp` (new method in an interface), the implementation in `Lit` (new method in a class), and a call in method `run()` of class `Test` (method extension) .

However, extra code is required to make a hyperslice declaratively complete so that it can be compiled. For instance, variable `value` that is introduced in feature `lp` is replicated in class `Lit` so that it can be returned by method `eval()`. Something similar occurs with variable `ltree` in `Test`. Additionally, the Hyper/J weaver requires stubs for non-default constructors. When the package is compiled, the references of these variables are bound to the definitions in the package; however, when composed with other hyperslices that also declare these variables, all the references are bound to a single declaration determined by the composition specification. The extension of methods and constructors is realized by appending the code of their bodies one after the other. The rest of the feature modules are implemented similarly.

The `LitAdd` composition is defined by the three files of Figure 4.5b: hyperspace `LitAdd.hs`, concern mapping `LitAdd.cm`, and hypermodule `LitAdd.hm`. The hyperspace file lists all the units that participate in the composition. The concern mapping divides the hyperspace into features (hyperslices) and gives them names. Finally, the hypermodule specifies what hyperslices are composed and what mechanisms (operators) to use. Our example merges units that have the same name.

```
interface Exp {
    int eval();
}
```

```
class Lit implements Exp {
  public int value;  // stub lp
  public Lit (int v) { } // req constructor
  public int eval() { return value; }
}
```

```
class Test {
  Lit ltree; // stub lp
  public void run() {
   System.out.println(ltree.eval());
  }
}
```

(a) Package LE of feature le

```
Hyperspace (hs)
hyperspace  LitAdd
 composable class LP.*;
 composable class LE.*;
 composable class AP.*;
 composable class AE.*;
```

```
Concern Mapping (cm)
package LP : Feature.LP
package LE : Feature.LE
package AP : Feature.AP
package AE : Feature.AE
```

```
Hypermodule  (hm)
hypermodule LitAdd
 hyperslices:
   Feature.LP,
   Feature.AP,
   Feature.LE,
   Feature.AE;
 relationships:
   mergeByName;
end hypermodule;
```

(b) Composition Specification

Figure 4.5 Hyper/J Implementation

4.4.2 Evaluation

Feature definition. Hyper/J's hyperslices can modularize all deltas, treat them as a cohesive unit, and compile them separately. Though, separate compilation requires manual completion of the hyperslices.

Feature composition. Hyper/J provides flexible composition. The order is specified in the hypermodule and can be done using several composition operators [151], thus composition order is flexible. Hyperslices are by definition closed under composition. To the best of our knowledge there is no theory to support static typing of hyperslices.

4.5 Jiazzi

Jiazzi [113][114][161] is a component system that implements units [64][65] in Java. A *unit* is a container of classes and interfaces. There are two types of units: *atoms*, built from Java programs, and *compounds* built from atoms and other compounds. Units are the modularization mechanism of Jiazzi. Therefore they are the focus of our evaluation.

4.5.1 Feature modules and their composition

Jiazzi programs use pure Java constructs. Jiazzi groups classes and interfaces in *packages* that are syntactically identical to Java packages. Implementation of modules like `lp` are thus standard Java packages with normal classes and interfaces. Consider the following code contained in package `le` that implements the feature of the same name[8]:

```
public interface Exp extends lp.Exp {
   int eval();
}
public class Lit extends lp.Lit
implements fixed.Exp {
   public int eval() { return value; }
}
public class Test extends lp.Test {
   public void run() {
     super.run();
     System.out.println(ltree.eval());
   }
}
```

Two important things to note are: a) `Exp`, `Lit` and `Test` *extend* their counterparts of feature `lp`, and b) class `Lit` implements `fixed.Exp` which refers to the version of `Exp` that contains all the extensions in a composition.

Package `le` shows how methods can be added to existing classes and interfaces, and how existing methods can be extended. Jiazzi also supports adding new classes, interfaces, constructor extensions, and fields in a similar way to that of normal Java inheritance. The rest of the feature modules are implemented along the lines of module `le`.

Composition in Jiazzi is elaborate. For simplicity, we illustrate unit composition with units `lp` and `le` instead of `LitAdd`. From this readers can infer what the composition of `LitAdd` entails.

8. Definition of non-default constructors is required but not shown.

We start with the definition of a *signature* which describes the structure of a package, i.e., the interface it exports. The following code is the signature of package `le`[9]:

```
signature leS = l : lpS + {
  package fixed;
  public interface Exp { int eval(); }
  public class Lit { public int eval(); }
  public class Test { public void run(); }
}
```

Two relevant points are: a) the expression `l:lpS +` indicates that `leS` is an extension of signature `lpS`, meaning that `Exp`, `Lit`, and `Test` of `le` extend their counterparts in `lp`, and b) `fixed` is a *package parameter* that is used, as we have seen, in the implementation of `le`. How this parameter is bound is explained shortly.

A unit definition consists of import and export packages followed (if necessary) by a series of statements that establish relations among the packages which, in the case of compound units, determines the order in which units are composed. Each of the features in our problem is implemented by an atom, and a program in the EPL is expressed by a compound unit. The following code defines unit `le`:

```
atom le {
  import lp : lpS;
  export le extends lp : leS;
  import fixed extends le;
}
```

It asserts that atom `le` imports package `lp` with signature `lpS` and that it exports package `le` of signature `leS` which is an extension of `lp`. It also states that it imports package `fixed`, an extension of `le` which is bound, at composition time, to the package parameter of the same name in the signature.

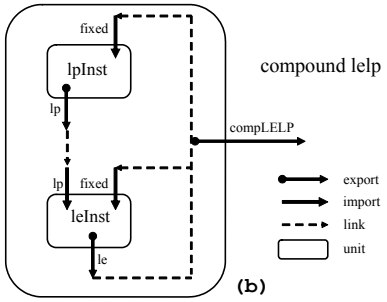
9. For convention in this section, we form signature names with the names of the packages they described followed by a S.

Jiazzi supports composition through the *Open Class Pattern* [113][114]. The key element of this pattern is the creation of a package, called `fixed` in our example, that contains all the extensions made by the units. This package is imported by the atom units, creating a feedback loop that permits them to refer to the most extended version of the classes and interfaces involved in a composition.

Figure 4.6a shows the code that composes these two units. Figure 4.6b illustrates this composition. Consider the second part of the specification first. It states that the composition contains two units (line 3): `lpInst` an instance of unit `lp`, and `leInst` an instance of unit `le`. The packages of these two units are linked as follows: a) line 4 states that the exported package `le` of `leInst` is bound to all the `fixed` packages in the compound, b) line 5 sets the link between the export package `lp` of `lpInst` to the import package `lp` of `leInst`.

```
compound lelp {
  export compLELP : leS;                    (1)
  bind package compLELP to compLELP@fixed; (2)
  }{
    link unit lpInst : lp, leInst : le;     (3)
    link package
        leInst@le to *@fixed,               (4)
        lpInst@lp to leInst@lp,             (5)
        leInst@le to compLELP;              (6)
}
```

(a)



(b)

Figure 4.6 Jiazzi Composition of le and lp

To be useful, compound packages must export something, in our case it exports a package that we named `compLELP` with signature `leS` (line 1) which is linked to package `le` of unit `leInst` in line 6. Since `compLELP` has signature `leS` that contains package parameter `fixed` we must bind it, in this case to itself, as done in line 2.

Signatures allow separate unit compilation. Jiazzi provides a stub generator that uses the unit's signature to create the packages and the code skeletons of the classes and interfaces required to compile the unit. It also provides a linker that checks that the compiled unit conforms to the unit's signature and stores the unit's binaries and signature into a Java archive (`jar`) file that can be used to compose with other units. For further details on the stub generator and linker refer to [115].

4.5.2 Evaluation

Feature definition. Jiazzi units can modularize all program deltas of EPL in a cohesive way. Furthermore, signatures allow separate compilation.

Feature composition. Jiazzi separates clearly the implementation of features from their composition thus provides a flexible composition. The order of unit composition is determined by the linking statements in compound units and therefore it is flexible. By definition, units are closed under composition. Jiazzi is backed up with a formal theory for type checking units and their compositions [64][65]. This theory permits the linker to statically check and report errors in program composition.

Jiazzi's type checking and separate compilation come with a price. Defining signatures and wiring the relationships between units is a non-trivial task, especially when dealing with multiple units with complex relations among them [161].

4.6 Scala

Scala is a strongly-typed language that fuses concepts from object-oriented programming and functional programming [143][126]. Though Scala borrows from Java, it is not an extension of it. We included Scala[10] in our evaluation because it supports two non-traditional modularization mechanisms: traits [142] and mixins [38].

4.6.1 Feature modules and their composition

A trait in Scala can be regarded as an abstract class without state and parameterized constructors. It can implement methods and contain inner classes and traits. We implemented each feature module by a trait. Consider the implementation of feature `lp` shown in Figure 4.7a. The trait contains:

- Abstract type `exp` with upper bound `Exp`. This means that `exp` is at least a subtype of `Exp` and thus it leaves `exp` open for further extensions by other features.

10. We used version 1.3.0.10 for our evaluation.

- Trait `Exp` declares method `print()`. A trait is used in this context because it is roughly equivalent to a Java interface, as it declares a type with methods whose implementations are not yet defined.
- Class `Lit` extends `Exp`.[11] It has a *primary constructor* (or main constructor) that receives an integer which is assigned to field `value`. It also provides an implementation for method `print()` that displays this field.
- Class `Test` contains abstract field `ltree` of abstract type `exp`. Because of this, class `Test` is also `abstract`. `Test` also contains method `run()` that calls method `print()` on `ltree`.

Trait `ap` is implemented as an extension of trait `lp`, shown in Figure 4.7b, that contains:

- Class `Add` that extends trait `Exp` of module `lp`. It has a two parameter constructor to initialize the expression fields and the implementation of method `print()`.
- Extension to class `Test`, that adds field `atree` and extends method `run()` with the call to `print()` on this field[12]. This class is also `abstract` because `atree`'s type is abstract.

Trait `le` is also implemented as an extension to trait `lp` and is shown in Figure 4.7c. This trait has:

- Trait `Exp` extends `Exp` of feature `lp` by adding method `eval()`.
- Abstract type `exp` that extends `exp` of feature `lp`, meaning that `exp` is now at least a subtype of `Exp` that has `print()` and `eval()` methods.
- An extension of class `Lit`. This class uses *mixin composition* (expressed as `with Exp` in the figure) to indicate that `Lit` is also a subtype of `Exp` and thus it must implement both of its methods. Since it inherits `print()` from trait `lp` it only needs to implement `eval()`.

---

11. Scala traits are conceptually not different from classes so that is why we use an extends clause instead of implements.
12. To prevent inadvertent overriding, Scala requires overriding methods to include an `override` modifier as part of their definitions. Notice also that the overridden method can still be called using super as in Java.

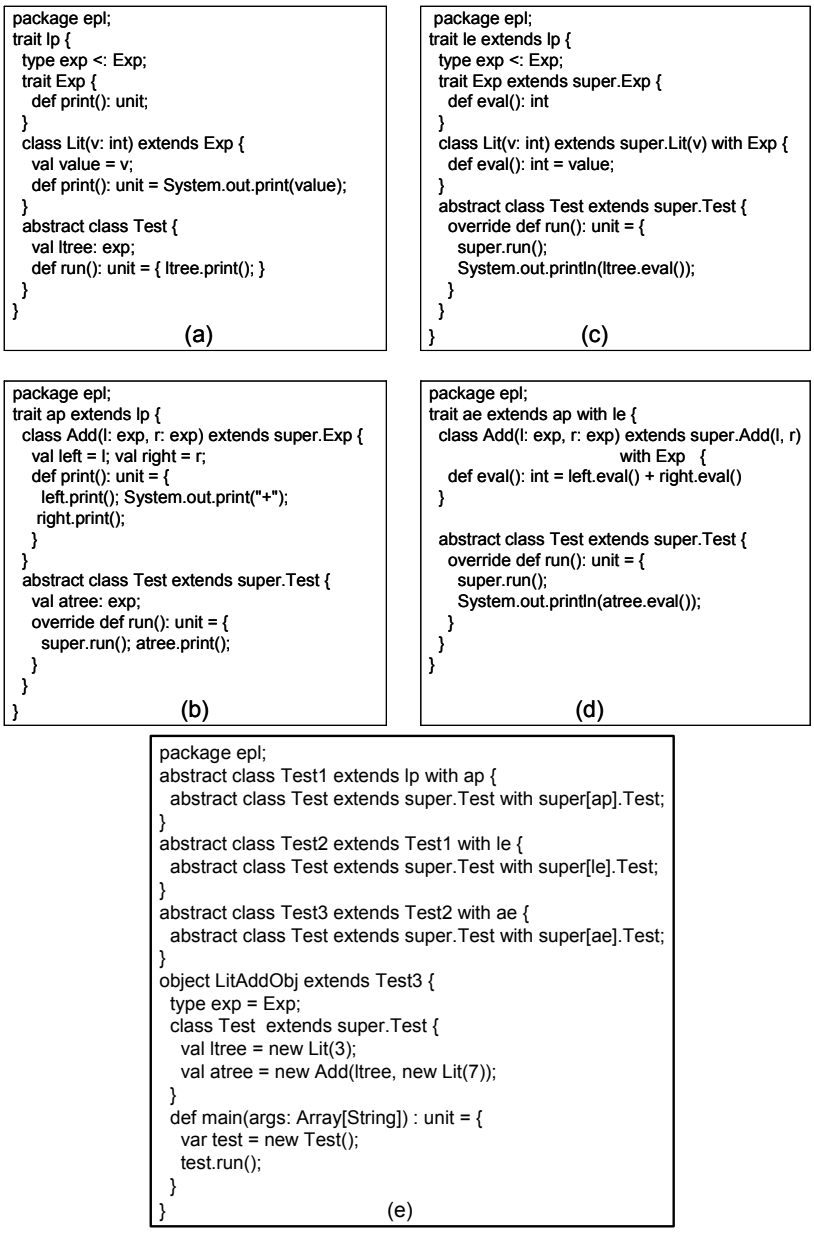


Figure 4.7 Scala Solution

- An extension of class `Test` that modifies `run()` to invoke `eval()` on `ltree`.

Feature `ae` is implemented as an extension of feature `ap` and a mixin composition with feature `le` because it provides an implementation of method `eval()` for class `Add`.

The code is shown in Figure 4.7d. Additionally this trait extends method `run()` of class `Test`. The other two feature modules of EPL, `np` and `ne`, are implemented similarly.

To define program `LitAdd` is necessary to: a) specify the order in which method extensions are composed, and b) to create an `object`, a singleton object of a new class, to run the program. Figure 4.7e illustrates this. For the first part, we use *deep mixin composition* [165] (mixin composition at trait level and nested class level), to establish a linear order of `Test` classes as they contain extensions of method `run()`. For the second part, we define `LitAddObj` that extends `Test3` (the most refined abstract `Test` class), binds abstract type `exp` to concrete type `Exp` as defined by `Test3`, and makes concrete class `Test` by creating instances for the test objects `ltree` and `atree`. The main method creates an instance of `Test` and calls method `run()` on it.

4.6.2 Evaluation

Feature definition. Scala can implement all program deltas of EPL. Regarding cohesion, traits provide a mechanism to collect program deltas under a single name. Separate compilation in Scala requires traits and classes to be placed in named packages, as it is illustrated by package `epl` in Figure 4.7.

Feature composition. Scala provides flexible composition and flexible order mechanism for implementing EPL. Scala uses inheritance and mixin composition to compose program deltas that add new classes, traits, fields, methods and simple constructor extensions. However, specifying the order of method extensions is a verbose and non-trivial task. Scala traits are closed under composition. Scala is supported by a sophisticated nominal type theory called vObj calculus [128].

4.7 CaesarJ

CaesarJ is an aspect oriented language that merges concepts of collaboration-based designs [147], mixins [38], aspects [12], and virtual classes [96] into modules called *class families* [119][120]. A goal of the language is to raise object oriented concepts such as late binding and subtype polymorphism to groups of classes.

4.7.1 Feature modules and their composition

Features modules and their classes are implemented with virtual classes, denoted with keyword `cclass`. For example, the following is the definition of feature `le`:

```
public cclass le extends lp {
      public cclass Expression {
            public int eval() { return 0; }
      }
      public cclass Lit {
            public int eval() { return value; }
      }
      public cclass Test {
            public void run() {
                  super.run();
                  System.out.println(ltree.eval());
            }
      }
}
```

CaesarJ utilizes mixin composition, denoted with operator `&`, to compose class hierarchies [8]. This operator is not commutative and implements a non-trivial variation of the C3 linearization algorithm to linearize mixin composition, for further details refer to [61][8]. The way the mixin linearization algorithm works can potentially permit many ways in which to specify a particular feature composition order. For instance, in our exam-

ple `LitAdd` features are composed in the order: `lp`, `ap`, `le`, and `ae`. This feature composition order can be obtained with 7 different mixin composition orders, one of which is:

```
public cclass LitAdd extends le & ap & ae & lp {}
```

The driver code creates an object of the desired product member and creates an instance of class `Test` to execute the method `run` on it as shown below:

```
public class Driver {
  public static void main (String[] args) {
      LitAdd ast = new LitAdd();
      LitAdd.Test test = ast.new Test();
      test.run();
  }
}
```

Among other functionality, CaesarJ also provides AspectJ-like advice and pointcut constructs. Thus it is possible to implement EPL with pointcuts and advice as shown in Section 4.3 and its evaluation would be similar to AspectJ's.

4.7.2 Evaluation

Feature definition. CaesarJ can implement all program deltas required for EPL. It provides support for feature cohesion as it groups classes within `cclass` modules; however, it does not provide separate compilation.

Feature composition. CaesarJ provides flexible composition and flexible order. However, determining the order in which to compose mixins with operator `&` to yield a particular composition order is a non-trivial task, specially when dealing with product line members with a significant number of features. CaesarJ's virtual classes are closed under composition. There is a calculus that describes a core of the virtual classes implemented in CaesarJ [61].

4.8 AHEAD

As we saw in Chapter 3 AHEAD is a feature modularization and composition technology based on step-wise development that was created to address the issues of feature-based development of product lines [30][27][1].

4.8.1 Feature modules and their composition

Recall that AHEAD partitions features into two categories: *constants* that modularize any number of classes and interfaces, and *functions* that modularize classes, interfaces and their extensions.

The implementation of constant features like `lp`, whose elements are standard classes and interfaces, uses pure Java constructs. Recall from Chapter 3 that Jak uses modifier keyword *refines* to distinguish class and interface refinements extensions. It also uses the construct `Super.methodName(args)` to refer to the method being extended. Thus, the Jak code of feature module `le`:

```
layer le;
refines interface Exp { int eval(); }

layer le;
refines class Lit implements Exp {
   public int eval() { return value; }
}

layer le;
refines class Test {
   public void run() {
      Super.run();
      System.out.println( ltree.eval() );
   }
}
```

As described in Figure 4.3, this feature extends interface `Exp` with method `eval()`, extends class `Lit` with the corresponding implementation, and extends class `Test` by extending method `run()` with a call to `eval()` on `ltree`. `Super.run()` invokes the previously defined method `run()`. In the case of `LitAdd` it calls the `run()` method of `ap`. Constructor extensions follow a similar pattern, as illustrated in the following example, which extends the constructor of `Test` of feature `ap` by assigning variable `atree` a value:

```
refines Test() {
   Add atree = new Add( ltree, ltree );
}
```

The remaining feature modules are implemented in a similar way. Each feature is represented by a directory that contains files for each class and interface definition and extension. The command line to compose these directories to form `LitAdd` is:

```
composer --target=LitAdd lp ap le ae
```

4.8.2 Evaluation

Feature definition. AHEAD can modularize all EPL program deltas into a cohesive unit. AHEAD provides tools to compile feature modules to bytecode and compose byte-code representations; however, this is not accomplished by separate compilation. Compilation uses global knowledge of all possible classes, interfaces, and members that can be present in a product-line [1].

Feature composition. AHEAD feature modules are independent of the composition. The order in which features are composed is the order in which they are listed on the *composer* command line. AHEAD features are by definition closed under composition. AHEAD currently does not have a static typing model.

4.9 Other Modularization Technologies

We evaluated feature modularity in several other modularization technologies. Our evaluation showed that they either do not support the program deltas required by EPL or they lacked appropriate tool support at the time of the evaluation. The following is a summary of the evaluation of these methodologies.

MultiJava. MultiJava is an extension of Java that supports symmetric multiple dispatch and modular open classes [44][46]. Its focus is on solving the *augmenting method problem*, that consists on adding operations (methods) to existing type hierarchies. Given this constraint, it is not possible to implement EPL as it cannot add new fields, add new classes and interfaces, and extend existing methods and constructors.

Classboxes. Classbox/J [34], the Java implementation of *Classboxes* [33], provides modules implemented as packages of classes that allow to add new classes and refine existing ones. A class refinement can new methods, new constructors and new fields. Classbox/J also supports a limited notion of method and constructor extension. The construct `original()` refers to the original definition of the method or constructor, in AHEAD terms the base definition. Packages are explicitly imported which reduces feature composition flexibility.

Difference-Based Modules. *Difference-Base Modules* is a modularization technology based on collaborations [78]. It can implement all the program deltas of EPL except constructor extensions. Its language, *MixJuice* [79], has constructs remarkably similar to AHEAD's with two major differences: a) module relations are hardwired (fixed), b) support for multiple inheritance (dealt with linearization and name-space policies). MixJuice compiler does not provide set true separate compilation. MixJuice modules are not closed under composition, and do not have a model of static typing.

Object Teams. *Object Teams* is a modularization technology also based on collaborations [74]. A *team* is a set of classes that implement the roles in a collaboration. It appears that the only program delta not supported by teams is constructor extensions. The relationships between teams are hardwired, and teams are not closed under composition.

Object Teams also has advice capability similar to AspectJ except that it only affects the execution of single methods (no name patterns). We are unaware of any model of static typing for Object Teams.

Aspectual Collaborations. *Aspectual Collaborations* is also a technology based on collaborations whose goal is to reconcile modules and aspects [97]. An a*spectual collaboration* consists of a *participant graph*, where nodes are Java-class like entities called participants, and the edges are *is-a* and *has-a* relations [132]. Participants do not have constructors, thus constructor extensions program deltas do not apply here. Participants have methods that can be defined as aspectual to advise the execution of other methods (consider them as methods that hold the advice code in AspectJ). The members of a participant can also have modifiers *expected* and *exported* to indicate the participants requirements with reference to other collaborations. Composition of collaborations is performed with *attach clauses*. Aspectual collaborations support separate compilation, provide flexible composition and order, are closed under composition, and have a model for static typing.

Delegation Layers. *Delegation Layers* is also another technology based on collaborations. Its goal is to scale Object Oriented mechanisms such as delegation, late binding, and subtype polymorphism to sets of collaborating objects [131]. The main differences with mixin layers [147] and AHEAD are: a) composition occurs at runtime, b) support of virtual classes [96] and family polymorphism [60]. Delegation layers appear to be able to implement all program deltas of EPL but it is unclear how they would support constructor extensions. Delegation layers do not have a flexible composition and order, and are not closed under composition. To the best of our knowledge, there is no model for static typing.

Aspectual Mixin Layers. *Aspectual Mixin Layers (AML)* capitalizes on the strengths of FOP and AOP [7]. In this context, features are mixin layers that contain mixin classes and aspects that can also refine pointcuts and pieces of advice. AML follows a function composition similar to that of AspectJ. A prototype tool is under development.

Classpects. *Classpects* are an attempt at unifying aspects and classes as units of modularization [137]. The authors argue that the asymmetric treatment of both concepts

makes understanding and using aspect oriented programming harder, complicates system composition, and harms modularity. Classpects enhance classes with *bindings* that pair an advice type and a pointcut with the method that implements the advice. Furthermore, classpects provide a more flexible mechanism for aspect instantiation and instance advising. Classpects do not provide support for feature cohesion, are not compiled separatedly, and are not closed under composition. We are unaware of any static typing model.

Open Modules. A common criticism of AOP is the unrestrictive capability of aspects to break standard modularization properties such as encapsulation and modular reasoning. *Open Modules* address this issue by providing a module system on top of aspects to specify methods and pointcuts to which client programs can apply advice [2]. A Java version of Open Modules has been recently developed [129]; because it is based on AspectJ, its evaluation for the implementation of EPL is equiparable to that in Section 4.3 with an improvement in feature cohesion.

Framed Aspects. *Framed aspects* merge frame technology and AOP [101][102]. AOP is used to modularize crosscutting concerns while frame technology is utilized for configuration, validation and variability via parameterization, conditional compilation and code generation. This approach relies on languages such as AspectJ and thus its evaluation is equiparable to that in Section 4.3.

4.10 Related Work

Expression Problem. The expression problem originated in the works of Reynolds [138] and Cook [48]. Torgersen [153] presents a concise summary of the research on this problem and four solutions that utilize Java generics. Though extensive, this literature focuses only on programming language design and separate compilation issues, and not about the requirements of feature modularity.

Comparison of AOP Models. Masuhara et al. describe a framework to model the crosscutting mechanisms of AspectJ and Hyper/J [112]. Both are viewed as weavers parameterized by two input programs plus additional information such as where, what,

and how new code is woven. Their focus is on the implementation of crosscutting semantics rather than on the broader software design implications that these mechanisms have.

The AOSD Network of Excellence published a survey of 28 AOP languages [15]. This survey makes a basic categorization of the AOP languages along two dimensions: a) Aspect Language dimension that compares characteristics such as types of join points, pointcuts, modules, and models of composition, weaving and instantiation; b) Execution Model dimension that includes characteristics such as implementation techniques and architectures (JVM, .NET, etc), runtime representation, and advice instantiation. The survey does not make a comparison based on standard problems and does not address the implications the different language mechanisms have on systems implementation let alone for product line development.

Comparison on Feature Refactoring. Murphy et al. [123] present a limited study that uses AspectJ and Hyper/J to refactor features in two existing programs. The emphasis was on the effect on the program's structure and on the refactoring process, not in providing a general framework for comparison. Coyler et al. [49] focus on refactoring tangled and scattered code into base code and aspects that could be considered as the features of a product line. They indicate that, based on their experience implementing middleware software, concerns (features) are usually a mixture of classes and aspects; a finding that corroborates the importance of feature cohesion. Liu et al. [99] describe an algebraic theory of feature refactoring that could be used to compare modularization technologies in terms of refacturing capability.

Comparison on Systems Implementation. Driver [58] describes a re-implementation of a web-based information system that uses Hyper/J and AspectJ, but the evaluation is subjective and expressed in terms of factors such as extensibility, plugability, productivity, or complexity. Clarke et al. [41] describe how to map crosscutting software designs expressed as composition patterns (extended UML models) to AspectJ and Hyper/J, and evaluate their crosscutting capabilities to implement such patterns.

Acknowledgements. We thank the following persons for their help with the implementation of EPL: Bin Xin and Sean McDirmid for Jiazzi, Martin Odersky for Scala, and Iris Groher for CaesarJ.

Chapter 5

Towards an Algebraic Model of Features

In Chapter 4 we analyzed several approaches to feature modules. To better compare and contrast them it is crucial to devise a model that abstracts implementation details and exposes the fundamentals. As a first step towards that goal, and inspired by the work on AHEAD, this chapter describes an algebraic model for the core of AspectJ. We chose to start with AspectJ because it is the most popular of the approaches we surveyed. An immediate benefit of this model was highlighting the source of some problems documented for this language. We propose an alternative model based on function composition to address these problems[1]. We relate these models with AHEAD's model in Chapter 3 and the properties of feature modules evaluated in Chapter 4.

5.1 Motivation

An aspect can be regarded as a declaration of changes that are to be made to a program; the process of making these changes is called *weaving*. There have been several proposals to define aspect semantics [56]. The most common uses an event-based model [157]. There are many ways in which aspects and weaving can be implemented. The historical roots of AOP are in meta-object protocols (MOPs) [40]. A MOP alters the metaclasses that control the behavior of a program's execution. On the other hand, aspect compilers perform static weaving by producing woven binaries or woven source [17]. When a

1. The core of this chapter comes from our PEM paper [108].

woven binary is run, its execution flow is indistinguishable from that of a MOP implementation. Aspect compilers are popular today because, among other reasons, they offer improved program run-time performance through static optimizations that are too expensive to realize via MOPs [18].

If we want to understand the impact that aspects have on a program's structure, we need to study an implementation of aspects that makes program structure explicit. We use program transformations for that purpose. A program transformation is a function that maps programs to programs [135]. While there are few aspect compilers [70] that explicitly use program transformation tools [145], the effects of weaving can be understood in terms of transformations. This connection enables us to raise aspects from code artifacts to mathematical entities (functions from programs to programs) and develop an algebraic model of aspects and their composition. This model reveals aspect composition as a significant source of some documented problems in AspectJ: limited reuse [72], hard to predict behavior [116], and difficult modular reasoning [43][2]. All these hinder useful software engineering practices such as step-wise development [159] and its natural materialization in *Component-Based Software Engineering (CBSE)* [149], where programs are developed incrementally by composing components one at a time.

5.2 Crosscuts as Program Transformations

Recall from Chapter 4 that AspectJ has two types of crosscuts, static and dynamic. This section describes how both can be interpreted as program transformations.

Static Crosscuts as Transformations. Static crosscuts affect the static structure of a program [11][93]. Our model focuses on introductions, also known as *inter-type declarations*, that add fields, methods, and constructors to existing classes and interfaces. Consider class `Point` defined below:

```
class Point {
  int x;
  void setX(int v) { x = v; }
}                                                    (22)
```

The following aspect `TwoD` adds (introduces) a second coordinate value to class `Point`. It adds field `y` and method `setY`:

```
aspect TwoD {
  int Point.y;
  void Point.setY(int v) { y = v; }
}
```

When these two files are composed or woven by the AspectJ compiler `ajc` using the command:

```
ajc Point.java TwoD.java
```

The result is a new class `Point'` with the introduced members underlined below:

```
class Point' {
  int x;
  void setX(int v) { x = v; }
  int y;
  void setY(int v) { y = v; }
}                                                              (23)
```

AspectJ generally uses more sophisticated rewrites than those shown here. The composed code snippets we present simplify illustration and are behaviorally equivalent to those produced by `ajc`.

From the program transformation perspective, base code such as `Point` in (22) represents a value to which a function (a program transformation) or aspect is applied. For instance, class `Point'` in (23) can be written as the following expression:

```
Point' = TwoD ( Point )
```

That is, `Point` is a base program and `TwoD` is a function that maps `Point` to `Point'`.

Dynamic Crosscuts as Transformations. Dynamic crosscuts, in contrast, run additional code when certain events occur during program execution. The following aspect is the familiar logging example. Its interpretation is: run the advice code (underlined) `after` (advice type) the execution of methods in class `Point` whose name starts with '`set`' (*pointcut in italics*).

```
aspect Logging {
  after():  execution(* Point.set*(..))
  { println("Logged"); }
}                                                    (24)
```

From a compiler perspective, an equivalent interpretation is: *insert* the advice code after the body of any method in class `Point` whose name starts with '`set`'. For example, if aspect `Logging` is woven into class `Point'` in (23) the result is equivalent to:

```
class Point" {
  int x;
  void setX(int v) {x = v; println("Logged"); }
  int y;
  void setY(int v){ y = v; println("Logged"); }
}                                                    (25)
```

Dynamic crosscuts can be implemented by transformations. For example, class `Point"` in (25) can be written as the expression:

```
Point" = Logging(Point') = Logging(TwoD(Point))
```

That is, class `Point"` is the result of applying two transformations, or from an AOP perspective the result of weaving two aspects, into class `Point`.

AspectJ provides an array of sophisticated mechanisms to define pointcuts and to perform complex rewrites when weaving aspects into programs. All dynamic crosscuts can be understood as transformations including pointcut designators such as `cflow`, `args`, `this`, and `target`, which expose context information of a join point [11].

Consider `cflow(Y)` where `Y` is a pointcut. Suppose `Y` captures a specific method execution or method call. `cflow(Y)` is the set of join points that occur during the execution of `Y`, from the time that the method is called to the time of the return [11]. An interesting question to ask is if a join point `X` occurs within the control flow of `Y`? The pointcut that expresses this is concisely written in AspectJ as:

```
cflow( Y ) && pointcut_for_X
```

From a compiler's perspective, control flow advice is a transformation that is composed from four simple transformations: (i) introduce a control flow stack `S`, (ii)

before each `Y` join point, push a marker `M` on `S`, and (iii) after each `Y` join point, pop `M` off `S`. For the duration that `M` is on `S`, any join point that occurs does so within the control flow of `Y`. And finally, (iv) at each `X` join point, check to see if `M` is on `S`; if so, execute the advice code.

Despite a significant body of research, regarding aspects as program transformations remains a controversial issue. We address this controversy in Section 5.7.1.

5.3 Advice Precedence

Recognizing that aspects can be realized as program transformations is a key first step in understanding how aspects impact program structure. The next step is to see how aspects are composed.

Multiple pieces of advice can be applied to the same join point. *Advice precedence* determines the order in which advice is woven. AspectJ deals with precedence differently depending on where the pieces of advice are defined, either in the same aspect or in different aspects [11].

Ordering aspects. AspectJ programmers have the option of declaring the order in which aspects are woven by a precedence statement such as:

```
declare precedence: Aspect3, Aspect2, Aspect1;
```

In the above example, the advice of $\texttt{Aspect}_1$ is woven first, then the advice of $\texttt{Aspect}_2$, and finally the advice of $\texttt{Aspect}_3$.[2] If no precedence statement is declared, the precedence of aspects is undefined in the semantics of AspectJ. In such cases, the AspectJ compiler chooses an order in which to weave aspects. In general, this order cannot be inferred by programmers prior to weaving.

Unfortunately, different weaving orders can result in programs that behave differently. Let us demonstrate with an analogy from arithmetic: given a value 2 and functions `double(x)`, `add3(x)`, and `sub2(x)` doubling the input, adding 3, and subtracting 2,

2. The mathematical concept of precedence has the opposite meaning of precedence in AspectJ. Higher precedence in AspectJ means apply later, whereas mathematical precedence means apply earlier.

respectively, different orders of their composition yields different results: `double(sub2(add3(2))=6` but `sub2(double(add3(2)))=8`. This is the reason why expressions, not sets of operations, are evaluated. In our setting, a value is a base program and the functions are aspects.

We need to know the weaving order to be able to predict the result. If the weaving order is undetermined, as in the absence of a precedence declaration, the woven program will at the least be not portable (since different compilers can choose different weaving orders). Moreover, programmers will not be informed about the order the compiler chooses, i.e., they will find it hard to predict the result of a weaving [13].

Ordering advice. Within an aspect, different pieces of advice appear in a certain textual order. However, the precedence of advice is governed by the following rules copied verbatim from [11]:

If two pieces of advice are defined in the same aspect, then there are two cases:

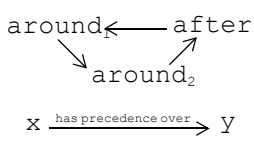
- If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

These precedence rules lead to two problems: 1) they may introduce a circularity such that the compiler cannot decide with which piece of advice to start the weaving, and 2) they cannot express all composition (weaving) orders.

The circularity problem is well-known [11], but the latter is not. A single aspect with three pieces of advice (identified by subscripts) illustrates both:

```
aspect Circular {
  void around1():execution(void test.main(..))
   { println("A1"); proceed(); println("A1"); }
  after3(): execution(void test.main(..))
   { println("A3"); }
  void around2():execution(void test.main(..))
```

```
    { println("A2"); proceed();println("A2"); }
}                                                    (26)
```



First, when the rules are applied, a circular precedence is created, as illustrated in the diagram to the right. To resolve the problem, programmers must manually modify the order in which the advice is listed in the program text, ensure that the resulting weaving order eliminates circularity, and produce a semantically appropriate weaving for the task at hand, a non-trivial and lengthy process.

Second, some composition orders cannot be attained. Suppose we want the following output sequence `(A2, A1, <main>, A1, A3, A2)`, which is achieved by weaving `around1` first, then `after3`, and then `around2`. In what order should advice $\texttt{around}_1$, $\texttt{after}_3$, and $\texttt{around}_2$ be listed in a single aspect file to achieve this weaving order?

The above rules dictate that `around2` must be listed before `around1` (because `around1` must be woven first). Advice `after3` must also appear before `around2` (to weave `after3` first). Thus, the ordering so far is: `after3` then `around2` then `around1`. But `after3` must also appear after `around1` (for `around1` to be woven before `after3`). It is impossible for `after3` to be *both* before `around2` and after `around1`. Thus, no linear ordering of the advice `around1`, `around2`, and `after3` can achieve the desired weaving order.

A way to realize such a weaving is to store each advice in a separate aspect file and use `declare precedence`:

`declare precedence:` $\texttt{around}_2$`,` $\texttt{after}_3$`,` $\texttt{around}_1$`;`

Another way might be to convert all after and before advice into around advice, which can be easily ordered. But this then begs the question of why after and before advice have different ordering rules than around advice.

In summary, the current rules for precedence makes program reasoning unnecessarily difficult. But precedence is not the only problem with aspect composition. Fundamental software engineering practices such as step-wise development are not satisfactorily supported by AspectJ, as the following section shows.

5.4 Incremental Development Example

Incremental or *Step-Wise Development (SWD)* is a fundamental programming practice [30][149][159]. It aims at building complex programs from simpler ones by progressively adding programmatic details. SWD is a centerpiece of core results in the synthesis of programs in product-lines [30] and component-based software engineering [149].

We illustrate a small but typical example of incremental development. Subscripts denote a particular version of our program at a given step and underline the code that is added by each increment.

Base. Class $\texttt{Point}_0$ defines a 1-dimensional point with an `x` coordinate and corresponding `setX` method:

```
class Point0 {
   int x;
   void setX(int v) { x = v; }
}                                                    (27)
```

First increment. Adds coordinate `y` and its `setY` method to $\texttt{Point}_0$ yielding:

```
class Point1 {
   int x;
   void setX(int v) { x = v; }
   int y;
   void setY(int v) { y = v; }
}
```

Second increment. Counts how many times the set methods are executed. Adding both increments to base results in:

```
class Point2 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
```

```
   int counter = 0;
}
```

Third increment. Adds a `color` field and `setColor` method to $\texttt{Point}_2$:

```
class Point3 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter = 0;
   int color;
   void setColor(int c) { color = c; }
}
```

Now here is an implementation in AspectJ:

Base. Identical to (27) because classes are the base code of AspectJ applications.

First increment. We define aspect `TwoD` that introduces field `y` and method `setY` to class `Point`:

```
aspect TwoD {
   int Point.y;
   void Point.setY(int v) { y = v; }
}
```

The command that composes class $\texttt{Point}_0$ and aspect `TwoD` and achieves a program equivalent to $\texttt{Point}_1$ is:

```
ajc Point0.java TwoD.java
```

Second increment. Aspect `Counter` introduces field `counter` to class `Point` and advises the execution of all set methods to increment this counter[3]:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : execution(* Point.set*(..))
```

3. There are other ways to define `Counter`. Shortly, we will present the rationale behind this implementation decision.

```
                    && target(p)
  { p.counter++; }
}                                                                    (28)
```

A program that is equivalent to $\texttt{Point}_2$ is produced by:

```
ajc Point0.java TwoD.java Counter.java
```

Third increment. Aspect `Color` adds a `color` field and a `setColor` method:

```
aspect Color {
 int Point.color;
 void Point.setColor(int c) { color = c; }
}
```

The composition of base with the three increments is:

```
ajc Point0.java TwoD.java Counter.java Color.java
```

However, this time the result is not $\texttt{Point}_3$, but instead:

```
class Point3' {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter;
   int color;
   void setColor(int c){ color=c; counter++; }
}
```

That is, the `setColor` method of $\texttt{Point}_3'$ increments `counter` (underlined above) unlike $\texttt{Point}_3$. This means that developers face the problem that building a program incrementally by hand may yield different results than using AspectJ. A more constrained version of `Counter`, that captures execution join points only of `setX` and `setY` methods, is required to produce `Point3`:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : (execution(* Point.setX(..))
```

```
            || execution(* Point.setY(..)))&& target(p)
    { p.counter++; }
}                                                          (29)
```

An obvious question is: why was `Counter` not defined like (29) in the first place? Doing that certainly would solve *this* problem, but we must consider other properties of software modules that are also desirable for aspects. Among them is reusability, i.e., we want to treat aspects as components as in CBSE and reuse them *as is*. For example, suppose `Counter` is redefined as (29), but now we want to build program $\texttt{Point}_3\texttt{'}$ instead. We would have to revise `Counter` back to (28) as the version in (29) cannot be used. We could conceivably keep in our code repository the two versions of `Counter` and select and apply one depending on the program we would like to synthesize. Regrettably, this alternative does not scale because it requires a specialized pointcut for each program that could be synthesized.The question now becomes: why can we not reuse the same aspect for both cases? The problem is that aspect weaving does *not* distinguish among development stages of a program. We show how to solve this problem in the next section.

5.5 An Algebraic Model of Aspects

In this section, we develop an algebraic model that reveals another source of complexity in AspectJ composition. We then propose an alternative model of composition that retains the power of AspectJ, supports step-wise development, simplifies advice precedence, and facilitates reasoning using aspects. Our model has three operations that build upon the notions of introduction, advice, and weaving.

5.5.1 Preliminaries

Our model requires all method introductions to be explicit. Advice in AspectJ implicitly introduces a method. Recall the `Logging` aspect:

```
aspect Logging {
  after(): execution(* Point.set*(..))
```

```
  { println("Logged"); }
}
```

When woven into class $\texttt{Point}_0$ in (22), aspect `Logging` can be regarded as the transformation that results in:

```
class Point  {
   int x;
   void setX(int v) { x = v; printLog(); }
   static void printLog(){ println("Logged");}
}
```

Method `printLog` is an explicit method that contains the advice body (the log message) and it is called at the end of the body of method `setX` (after the method execution).

Pure advice is a named advice that separates advice from introductions and replaces the advice body with a method call. To preserve AspectJ semantics, this call is not advisable, i.e., it has no join points. All join points of the original advice body reside in the method that is called. For example, we can conceptually rewrite the `Logging` aspect as:

```
aspect Logging {
  static void Point.printLog()
   { println("Logged"); }

  Log is after():execution(* Point.set*(..))
        --> Point.printLog();
}                                                              (30)
```

where `Log` is the name given to the pure advice, `printLog` is the method that contains the advice body, and the `-->` arrow indicates the method call. Other researchers have advocated a similar concept (that of making advice bodies into explicit methods), but have motivated the idea for different reasons [137]. Without loss of generality, we assume all advice is pure advice in the remaining of this chapter.

5.5.2 Introduction Sum

An *introduction* is a function that adds a data member or method to a program. Recall aspect `TwoD` and class $\texttt{Point}_0$ whose composition was modeled algebraically as:

$$\texttt{Point}_1 = \texttt{TwoD}(\texttt{Point}_0) \tag{31}$$

where $\texttt{Point}_0$ and $\texttt{Point}_1$ are values, and `TwoD` is a function that maps class $\texttt{Point}_0$ to class $\texttt{Point}_1$. Appealing to intuition, we can rewrite (31) as the sum of the introductions of `TwoD` with $\texttt{Point}_0$:

$$\texttt{Point}_1 = \texttt{TwoD} + \texttt{Point}_0 \tag{32}$$

Operation + is called *introduction sum*. It is a binary operation that performs disjoint union on *program fragments*, which are sets of variables and methods. For example, aspect `TwoD` is the program fragment (set) containing `y` and `setY`, and class `Point0` is the fragment (set) containing `x` and `setX`. We write introduction sum as:

$$\begin{aligned}\texttt{Point}_1 &= \texttt{TwoD} + \texttt{Point}_0 \\ &= \texttt{(setY + y) + (setX + x)} \\ &= \texttt{setY + y + setX + x}\end{aligned}$$

meaning $\texttt{Point}_1 = \{\texttt{setY, y, setX, x}\}$ where our notation above omits set brackets. As + is disjoint set union, introduction sum has the following properties:

Identity. `0` is the *empty program* (i.e., a program fragment that contains no members). If `X` is a program fragment:

`X = X + 0 = 0 + X`

Commutativity. + is commutative because set union is commutative.

Associativity. + is associative as set union is associative.

Operation + differs from AspectJ introduction in that it allows adding new classes and interfaces but does not allow member overriding.

5.5.3 Weaving

Pure advice is a function that maps an input program to a program where calls to advice methods have been inserted. We use function application to model the operation of *weaving*. Let `a` be pure advice and `P` be a program. The result of applying (weaving) `a` into `P` is program `P'`:

```
P' = a(P)
```

Weaving has the following properties:

Identity. `id` is the *null pure advice* — i.e., pure advice that captures no join points. Null pure advice is the identity transformation; its application does not affect a program. If `P` is a program fragment, `P=id(P)`. That is, `P` does not change when woven with `id`.

Fixed Point. If pure advice `a` is woven to program fragment `P` but a does not capture any join points of `P`, the outcome of the weaving is `P`. That is,

```
P = a(P)
```

Associativity. Weaving is right associative.

Distributivity. Weaving distributes over introduction sum. Let `P` be a program, `a` be pure advice, and `P'=a(P)`. Suppose `P=X+Y+Z`, where `X`, `Y`, and `Z` are arbitrary program fragments. We have:

```
P'    = a(P)
      = a(X + Y + Z)
      = a(X) + a(Y) + a(Z)
```

Advice applies to *all* join points in a program. Thus it is immaterial if the program fragment is viewed as a whole (`P`) or as the sum of its parts (`X+Y+Z`). This distributivity property is central to AOP.

Partial Distributivity. Corollary of Distributivity.

```
P'    = a(P)
      = a(X + Y + Z)
      = a(X) + a(Y + Z)
```

5.5.4 Advice Sum

Each piece of advice is a transformation (i.e., a function). The application of multiple pieces of advice is modeled by function composition, denoted by •, which we call *advice sum*. • also models advice precedence. `a3•a1` means apply advice `a1` first and then `a3`. Advice sum has the properties:

Identity. `id` is the null pure advice. If `a` is pure advice:

`a = a • id = id • a`

Commutativity. The order in which advice is applied matters. • is not commutative.[4]

Associativity. • is associative because function composition is associative.

5.5.5 Modeling Aspects as Pairs

We model an aspect as a pair of two entries. The first entry, called the *advice part*, is the aspect's advice and the second entry, called the *introduction part*, is the aspect's introductions. The pair for `Logging` (30) is:

`Logging = <Log, printLog>`

where `Log` is the name of the pure advice and `printLog` is the name of the introduced method.

`Counter` is another example. A pure advice version of it is:

```
aspect Counter {
   CounterP is after(Point p):
     execution(* Point.set*(..)) && target(p)
     --> Point.counterInc(p);

   static void Point.counterInc(Point p)
     { p.counter++; }
```

4. Two pieces of advice commute if they have no join point in common.

```
   int Point.counter = 0;
}
```

and its pair is:

```
Counter = <CounterP, counterInc + counter>
```

Note that the second entry of the pair sums the introductions `counterInc` (the method of the advice body) and `counter` (the variable).

Finally, a pure version of the `Circular` aspect (26) is:

```
aspect Circular {
   pointcut pcd() : execution(void test.main(..));
   static void test.m1(){ ... }
   static void test.m3(){ ... }
   static void test.m2(){ ... }
   a1 is void around1(): pcd() --> test.m1();
   a3 is after3():       pcd() --> test.m3();
   a2 is void around2(): pcd() --> test.m2();
}
```

Suppose advice is woven in the textual order listed in an aspect. `Circular` would be modeled by the pair:

```
Circular = <a2 • a3 • a1, m2 + m3 + m1>
```

The expression `a2•a3•a1` represents compound advice, where `a1` is woven first and `a2` last.

5.5.6 Pair Model of Aspect Composition

Let aspects `A1` and `A2` be modeled by the pairs $A1=\langle a_1, i1\rangle$ and $A2=\langle a2, i2\rangle$. We denote aspect composition in AspectJ by operation ◊. The AspectJ composition of `A2` with `A1` (with `A1` being applied first) is:

$$A2 \lozenge A1 = \langle a_2, i_2\rangle \lozenge \langle a_1, i_1\rangle = \langle a_2 \bullet a_1,\ i_2 + i_1\rangle$$

◊ is similar to vector addition because the coordinates of pairs are summed: + sums program fragments, • sums advice in weaving order.

As another example, program P can be modeled by the pair <id,p>, where id is null pure advice and p is the introduction sum of the members of P. Weaving aspect A1 into P and then weaving aspect A2 is:

$$\begin{aligned} &A2 \lozenge A1 \lozenge P \\ &\quad = \langle a_2, i_2 \rangle \lozenge \langle a_1, i_1 \rangle \lozenge \langle id, p \rangle \\ &\quad = \langle a_2 \bullet a_1 \bullet id,\ i_2 + i_1 + p \rangle \\ &\quad = \langle a_2 \bullet a_1,\ i_2 + i_1 + p \rangle \end{aligned} \tag{33}$$

The *code* of a pair is the program that the pair represents. Let V be a pair. Its code, denoted [V], is computed by weaving its advice part with its introduction part:

$$[V] = [\langle a, i \rangle] = a(i) \tag{34}$$

This follows from the fact that advice can advise any join point of a program[5]. Thus the program that is produced by weaving A1 and then A2 into P is:

$$[A2 \lozenge A1 \lozenge P] = a_2 \bullet a_1(i_2 + i_1 + p) \tag{35}$$

More generally, the result of weaving into P aspects $A_1 \ldots A_n$ (in this order) is:

$$\begin{aligned} &[A_n \lozenge A_{n-1} \lozenge \ldots \lozenge A_1 \lozenge P] \\ &\quad = a_n \bullet a_{n-1} \bullet \ldots a_1(i_n + i_{n-1} + \ldots + i_1 + p) \end{aligned} \tag{36}$$

That is, the result of weaving a sequence of aspects into a program equals the weaving of advice in weaving order into the program that is the introduction sum of the program's members and aspect introductions. (36) represents the "shape" of any program produced by AspectJ. We call this the *pair model* of composition.

(36) identifies the source of the problems noted earlier in Section 5.4 about incremental program development using AspectJ. It can be seen in the expansion of (35):

$$\begin{aligned} &[A2 \lozenge A1 \lozenge P] \\ &\quad = a_2 \bullet a_1(i_2 + i_1 + p) \\ &\quad = a_2 \bullet \underline{a_1}(i_2) + a_2 \bullet a_1(i) + a_2 \bullet a_1(p) \end{aligned}$$

5. Readers familiar with advice that advises itself will recognize that the pure advice part advises its introduction part. This is modeled by (34).

The offending term is underlined. It means that to apply aspect A2, the programmer is required to know how an advice from a previous development step (a_1) affects an introduction added in the current step (i_2). More generally, the weavings that cause problems in incremental development are underlined below:

$$\ldots \bullet a_{k+2} \bullet a_{k+1} \bullet a_k \bullet \underline{a_{k-1}} \bullet \underline{\ldots} \bullet \underline{a_2} \bullet \underline{a_1}(i_k) + \ldots$$

In other words, a programmer needs to know how previously applied pieces of pure advice a_j affect later introductions i_k where $j < k$. These are the terms that make step-wise development difficult. The problem is aggravated when a large number of aspects are composed and the development involves multiple steps.

5.5.7 Functional Model of Aspect Composition

In Chapter 3 we described the role of function composition in the synthesis of product lines [28][29]. We propose an alternative model, that equates aspect composition with function composition and eliminates the problems mentioned while preserving the power of AspectJ. Consider aspect A=<a,i>. We can model A as the function:

$$A(x) = a(i + x)$$

That is, A adds its introductions (i) to its program fragment input (x) before weaving its advice (a). So applying aspect A1 to program P and then applying aspect A2 is:

$$\begin{aligned} A2(A1(P)) &= a_2(i_2 + a_1(i_1 + p)) \\ &= a_2(i_2) + a_2 \bullet a_1(i_1) + a_2 \bullet a_1(p) \end{aligned} \quad (37)$$

Note that the offending pure advice a1 disappears from the left summand $(a_2(i_2))$. This generalizes to the composition of any number of aspects: the weavings that make step-wise development difficult are never generated. We call this the *functional model*. In effect, the pair model expresses *unbounded* quantification (i.e., the scope of advice extends over the entire program) [63], whereas the functional model expresses *bounded* quantification (i.e., the scope of advice extends over a stage in a program's development). An obvious question arises: which model is more expressive?

The functional model can express programs that the pair model cannot express, for example program (37). This expression cannot be produced by the pair model because all pieces of advice are woven into all introductions yielding program (35). Again, this is a consequence of the unbounded quantification of the pair model.

Unbounded quantification is a special case of bounded quantification. Think of quantifiers in first-order logic: the scope of a quantifier extends from its position in a logical formula to the right. Unbounded quantification means that the formula starts with the quantifier. In the functional model, a weaving expression is interpreted the same way: the influence of advice extends to the right and not to the left. Let us illustrate this with program (35) which requires unbounded quantification. To construct this program in the functional model, we decompose aspect `A1` into a pair of aspects, one containing introduction i_1 and the other with advice a_1:

$A1_{intro}(x) = i_1 + x$

$A1_{advice}(x) = a_1(x)$

Similarly for aspect `A2`:

$A2_{intro}(x) = i_2 + x$

$A2_{advice}(x) = a_2(x)$

To build (35), we weave the aspects with introductions to `P` first and then the aspects with advice to get:

$A2_{advice}(A1_{advice}(A2_{intro}(A1_{intro}(P))))$

$= a_2 \bullet a_1(i_2 + i_1 + p)$

This ability to model unbounded quantification in the functional model is general, and is not specific to this particular example.

In summary, the functional model can express all programs that the pair model can express, and more once aspects are expressed in terms of bounded quantification. Recall the `Point` example of Section 5.4. Let us add a third dimension to `Point`, which is defined by aspect `ThreeD` that introduces a `z` variable and `setZ` method. Assuming that `Counter` advises all set methods as in (28) using bounded quantification, we can build at least four programs:

(a) Color(ThreeD(TwoD(Counter(Point$_0$))))

(b) Color(ThreeD(Counter(TwoD(Point$_0$))))

(c) Color(Counter(ThreeD(TwoD(Point$_0$))))

(d) Counter(Color(ThreeD(TwoD(Point$_0$))))

Program (a) is a program that counts the executions of setX. (b) counts the executions of setX and setY. (c) counts the executions of setX, setY, and setZ. (d) counts the execution of all set methods. Each of these programs is synthesized by reusing and composing aspects *as is*. Using AspectJ, weaving aspects Counter, Color, ThreeD, and TwoD in an arbitrary order into P will always produce program (d). To build all four programs using AspectJ would require four different versions of Counter.

To summarize, problems in step-wise development arise using AspectJ when pointcuts are not bounded to a set of classes, methods, and variables at a specific stage of program development. Common examples are pointcuts that capture the set of all calls to one or more methods, and wildcard patterns. Subsequent introductions that are captured by these pointcuts give rise to the problems discussed here. The functional model avoids these problems.

5.6 Significance of Functional Model

Our work eliminates several problems in AspectJ. First, the precedence rules for ordering pieces of advice within an aspect (Section 5.3) can be eliminated. We propose a simpler rule: apply advice in the order in which it is listed in an aspect file. This rule will simplify the ordering algorithms currently utilized by aspect compilers and will help AspectJ programmers by reducing the effort to determine a composition order.

Second, the rules that AspectJ uses to assign precedence to aspect files can also be eliminated. We propose that a precedence be declared for *all* aspect files to define their composition order. Alternatively, the compiler could raise an error when users fail to specify an order where ordering matters. Again, this change simplifies advice ordering algo-

rithms for multiple aspect files and also helps programmers as now aspect compiler output will be predictable.

AOP researchers have raised the issue that it should be unnecessary to specify a composition order when aspects are provably commutative. We agree. In cases where pointcuts have disjoint sets of join points their corresponding advice is commutative. Existing tool support can help identify these situations [11]. However, it is still necessary to specify *when* these pieces of advice are to be applied.

5.7 Controversial Issues in AOP

There are several controversial issues around Aspect Oriented Programming. In this section we address the two most closely relate to our work.

5.7.1 Aspects as Program Transformations

Aspect compilers, such as `ajc`, demonstrate that aspects can be implemented by transformations: `ajc` takes a base program and aspects as input and produces a woven binary as output. Even so, the connection of dynamic crosscuts, especially `cflow`, to transformations remains controversial [73]. However, when given proper consideration, optimization and weaving techniques such as those presented in [18][75][112] are examples of program transformations, sophisticated indeed, but transformations nonetheless.

The relationship between program transformations and aspects is not new. Lämmel studied the implementation of aspects as programs transformations [94]. Kniesel et al. developed `JMangler`, a backend tool to support AOP that relies in transformations at the bytecode level [90]. The work of Krishnamurthi et al. on modular verification of aspect advice has an underlying assumption that aspects can be regarded as transformations [92]. We extend these ideas by showing how a transformation view leads to an algebraic model of aspect composition.

The history of automated software design is replete with results on transformations and their connection to program structure. Tool-enabled program refactorings are

program transformations [152]. Layers in layered software designs are transformations [29]. When viewed as increments in program functionality, features in software product-lines are transformations [30]. Model transformations play a key role in Model Driven Architectures; they are mappings between models (which are non-code representations of programs) [37]. Arguably the most significant result in automated software design is relational query processing [144]: a query evaluation program is defined by a composition of relational algebra operations. These operations are transformations of query evaluation programs.

Aspects define very useful transformations, and their strength is that programmers do not need understand transformation technologies to use them. As mentioned earlier, transformations are just one of a number of ways in which aspects can be implemented. The advantage of viewing aspects as transformations is that it exposes how aspects modify a program's structure. Doing so places aspects in context with results in software architectures and automated software development that also define and modify program structure.

5.7.2 Aspects and Modular Reasoning

Modular reasoning with AOP is controversial [44][45][46]. Kiczales and Mezini claim that in the presence of aspects "the complete interface of a module can only be determined once the complete configuration of modules in the system is known" [89]. In other words, aspects entail global reasoning that they define as "having to examine all the modules in the system or subsystems". The pair model of AspectJ mathematically corroborates their claim and shows the negative implications it has for incremental development. The functional model of composition reduces the need of global reasoning without restricting the power of AspectJ.

Rinard et al. propose a framework to classify aspects based on their interactions with other aspects and base code [139]. They present a tool that alerts users of cases where modular reasoning (a user-defined property) could be compromised so that users can take corrective action when necessary. *Open modules* proposes a module system whereby an

interface describes the pointcuts and join points that are advisable by the pieces of advice of other modules thus promoting modular reasoning about aspects [2].

5.8 Relating AspectJ and AHEAD Models

The composition of AHEAD in Chapter 3 has only one operation, function composition denoted by the symbol •. On the other hand, our models of AspectJ use three operations: i) introduction sum denoted by +, ii) advice weaving denoted by function composition, and iii) advice sum (unfortunately) denoted also by symbol •.

Developing both AspectJ models helped us realize that AHEAD operation • is overloaded; it can be interpreted either as introduction sum +, function application, or as function composition. This overloading did not represent a problem in the implementation of ATS. However, the integration of aspects into ATS must make these finer distinctions into account, and consequently it entails major changes to AHEAD composition model and tools. Therefore, the generalization of AHEAD model to include aspect operations and the corresponding implementation of supporting tools is part of our future work.

5.9 Algebraic Models and Feature Properties

In Chapter 4 we presented the basic properties of feature modules required for the implementation of our product line EPL. Our evaluation in that chapter illustrated how basic properties of feature modules can be abstracted from different modularization technologies. In this section we go a step further by relating those properties to our algebraic models. We believe this relation can constitute a foundation for a framework or set of criteria that a rigorous mathematical presentation should satisfy and can help reorient the focus on clean and mathematically justifiable abstractions when developing new tool-specific concepts.

In Chapter 4 we divided feature properties into definition and composition properties. Next we analyze how these properties can be expressed algebraically. The feature definition properties are:

- **Program deltas**. Program deltas were defined as program fragments, thus they are indeed increments in program functionality or features. In Chapter 3 we illustrated how features can be denoted by algebraic terms. The features or program deltas that we identified for EPL have the characteristic that cannot be divided further.
- **Cohesion**. Cohesion was defined as the capability of collecting, identifying, and manipulating sets of program deltas as a unit. In Chapter 3 we defined a feature module as a set of nested modules. Thus by collecting the constituents of feature models into sets with a name, we can refer to them as a cohesive unit. For example, feature `lp` in Figure 4.3 can be denoted as `lp={Exp,Lit,Test}`.
- **Separate compilation**. Separate compilation is not a property of an algebraic model, but rather a desirable engineering requirement of an implementation of our algebraic model.

Feature composition properties are:

- **Flexible composition**. AHEAD model explicitly separates feature modules, denoted by terms, from their composition denoted by expressions that operate on feature terms.
- **Flexible order**. We saw in Section 3.4 how Origami allows us to devise multiple composition expressions from a set a features and constraints.
- **Closure under composition**. Closure under composition follows from our definition of feature modules whereby a module can be formed with nested modules.
- **Static typing**. This property could be modeled as a new operation on feature modules.

5.10 Related Work

Since the Sixties, the paradigm of layered software has been applied to harness the complexity of large software systems (initially, they were operating systems [53]). A key property of layered software is that a layer has knowledge of lower layers but not of higher layers. (This is bounded quantification). We apply this principle to aspect orientation: the weaving of an advice affects the existing program, but not any parts that are added later.

Compositional models of aspects have a long history. GenVoca, AHEAD, and HyperJ are examples [29][30][63][150]. Relating compositional models to algebras is discussed in [30].

McEachen and Alexander consider the problems caused by weaving bytecode that already contains woven aspects [116]. A *foreign aspect* is an aspect that has been woven; the woven bytecode is later imported by a third party that has no access to the aspect's source code. A foreign aspect "comes alive" and can potentially affect subsequently added base or aspect code. Foreign aspects are problematic as they can: a) not capture all intended join points, b) capture unintended join points, and c) inadvertently interact with other aspects. The authors advocate guidelines to design the scope of pointcuts, use of abstract pointcuts to control the set of join points advisable by foreign aspects, and promote adequate pointcut documentation. Our work provides a foundation to understand the problems caused by foreign aspects and a solution to eliminate them. The rules of the functional model can be enforced by a compiler, whereas adhering to the guidelines of McEachen and Alexander is the responsibility of programmers.

Complementary to our approach, there is work that aims to improve aspect reuse by making significant language changes. Gybels and Brichau propose a logic-based crosscut language to better decouple aspects from programs [72]. Rho and Kniesel propose aspect uniform genericity, application of logic metavariables in language constructs, as a way to promote reuse and to significantly expand the generic capabilities of AspectJ [140]. Incidentally, they too take a transformation view of aspects.

Classpects unify aspects and classes [137]; they are classes enhanced with bindings. A *binding* associates an advice type (`before`, `after`, `around`) and a pointcut with a call to a list of methods. These methods replace the advice body, similar to what we did when we transformed advice into pure advice.

Chapter 6

An Assessment of the Functional Model

In Chapter 5 we presented the Pair Model and the Functional Model of aspect composition. In this chapter we show how to emulate the Functional Model (using the default Pair Model as implemented in `ajc`) for the implementation of a non-trivial product line consisting of five core tools of the AHEAD tool suite.

The emulation was achieved through a careful use of precedence clauses and a selection of language constructs that prevents the problems shown for the Pair Model and permits the composition specification defined by the Functional Mode. The emulation was tailored to the implementation of the five AHEAD tools and, as can be inferred from Chapter 5, is by no means a general mechanism to implement the Functional Model.

This case study serves several purposes: a) It helps gauge the potential benefits of the Functional Model, b) It shows how aspects can be used to implement non-trivial product lines of scale and complexity comparable to those implemented with AHEAD[1], c) It profiles the role different aspect constructs play in the synthesis of product lines, and d) It presents venues of research on the use of aspects in product line implementations.

We start by presenting a simple product line example to illustrate the kinds of features, composition, and mapping issues encountered in the translation of the AHEAD tools to AspectJ.

1. To the best of our knowledge, product lines implemented with AspectJ have been of smaller scale LOC or number of features.

6.1 Quadrilaterals Product Line

We start by developing a simple example, the *Quadrilaterals Product Line (QPL)*, to describe the types of features, composition, and mapping issues we encountered in translating the AHEAD tool suite to AspectJ.

The task at hand is to incrementally build two kinds of quadrilaterals: rectangles and trapezoids. Quadrilaterals have a `draw` operation that displays the lines between the four points that form a quadrilateral. However, rectangles and trapezoids use different lines, styles, and colors. Our example consists of three features.

Base feature. This feature defines: a) class `Quadrilateral` that contains four points and a `draw` operation that draws the lines between the points, b) class `Rectangle` that extends `Quadrilateral`, and c) class `Trapezoid` that extends `Quadrilateral` and overrides method `draw` to add new instructions to make the lines thicker. Feature `Base` can be implemented as follows:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}
class Rectangle extends Quadrilateral {}
class Trapezoid extends Quadrilateral {
   void draw() {
     super.draw(); ... thick lines ...
   }
}                                                              (38)
```

For sake of simplicity we omit details that differentiate rectangles and trapezoids and focus only on the `draw` methods.

Style Feature. This feature adds a pattern to fill rectangles and replaces operation `draw` in trapezoids to display dashed lines only. The result of composing features `Base` and `Style` is shown below, where the code that is added by the `Style` feature is underlined:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}
class Rectangle extends Quadrilateral {
  void draw() { super.draw();… fill pattern … }
}
class Trapezoid extends Quadrilateral {
   void draw() { ... dashed lines ... }
}                                                              (39)
```

Recall from Chapter 3 that a *class refinement* is the modularization of changes made by a feature to a class defined in another feature. Feature `Style` consists of two class refinements, one for `Rectangle` and one for `Trapezoid`. The class refinement of `Rectangle` overrides method `draw` defined in `Quadrilateral`. We refer to this type of refinement as a *method override*. The class refinement of `Trapezoid` effectively replaces method `draw` defined for this class in feature `Base`. We refer to this type of refinement as a *method replacement*.

Color Feature. `Color` adds a `color` field and a `setColor` method to `Quadrilateral`. It sets the color of rectangles to green and the color or trapezoids to blue.

The result of composing features `Base` with `Style` and `Color`, in that order, yields the following result where the changes caused by `Color` to (39) are underlined:

```
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
   int color;
   void setColor(int c) { color = c; }
}
class Rectangle extends Quadrilateral {
  void draw() { setColor(GREEN);
     super.draw();… fill pattern …
  }
```

```
}
class Trapezoid extends Quadrilateral {
   void draw() { setColor(BLUE);
    ... dashed lines ...
   }
}                                                                    (40)
```

The `Color` feature is implemented by three class refinements. The refinement of `Quadrilateral` adds `color` and `setColor`. The refinement of `Rectangle` extends the functionality of method `draw` by adding a call to `setColor` with value `GREEN`. Similarly, the `Trapezoid` refinement extends functionality of method `draw` with a call to `setColor` with value `BLUE`. We refer to these last two refinements as a *method extension*. We call the *derived method body* the part of the method extension that it is the result of the composition of all previous class refinements. In (40) this corresponds to the code of method `draw` that is not underlined.

Product Line Members. Based on the description of our three features, QPL can synthesize the following three products: 1) `Base`, 2) `Base` and `Style`, 3) `Base`, `Style` and `Color`. Though completely reasonable, QPL does not contain program with features `Color` and `Base`. Section 6.3.1 explains an interesting characteristic of this product in the AspectJ implementation of QPL.

Modularization Issues. We have seen in Chapter 4 that preprocessors, although commonly used and simple, are not first class modularization techniques; preprocessors, by definition, express concepts that are not supported by a host language. In the next two sections we evaluate AHEAD and AspectJ implementations of QPL.

6.2 AHEAD Implementation of QPL

Recall from Chapter 3 that AHEAD divides features into *constant* and *function* features. QPL is implemented with a constant feature for `Base`, and two function features, one for `Style` and one for `Color`. Next we present their implementation.

Base feature. `Base` is a constant feature that has three Jak files (one for each class):

```
layer Base;
class Quadrilateral {
   Point p1, p2, p3, p4;
   void draw() {... std lines ...}
}

layer Base;
class Rectangle extends Quadrilateral {}

layer Base;
class Trapezoid extends Quadrilateral {
   void draw() {
     Super.draw(); ... thick lines ...
   }
}                                                                    (41)
```

There are only two differences with the code in (38): the `layer` construct declares that these classes belong to feature `Base`, and Jak's keyword `Super` refers to the superclass of `Trapezoid`.

Style Feature. `Style` is a function feature that modularizes two class refinements (recognized by the keyword `refines`). In the `Rectangle` refinement, modifier `overrides` denotes that method `draw` is overridden and the superclass reference (`super`) in (39) is translated as `Super`. In the `Trapezoid` refinement, modifier `new` means that the `draw` method replaces the one defined for that class in feature `Base`.

```
layer Style;
refines class Rectangle {
  overrides void draw() {
      Super.draw(); … fill pattern …
  }
}
```

```
layer Style;
refines class Trapezoid {
   new void draw() { … dashed lines … }
}                                                                    (42)
```

Color Feature. `Color` is a function feature that modularizes three class refinements. Class refinement `Quadrilateral` adds new members `color` and `setColor`. Class refinements `Rectangle` and `Trapezoid` extend their `draw` methods by setting their corresponding colors and calling the `draw` method using keyword `Super`, underlined in the code below. One way to think about `Super` in method extensions is as a placeholder for the derived method body mentioned in (40).

```
layer Color;
refines class Quadrilateral {
   int color;
   void setColor(int c) { color = c; }
}

layer Color;
refines class Rectangle {
   void draw(){ setColor(GREEN); Super.draw();}
}

layer Color;
refines class Trapezoid {
   void draw() { setColor(BLUE); Super.draw();}
}                                                                    (43)
```

Product Line Members. AHEAD can generate the three programs of QPL using its `composer` tool. For instance the program with features `Base`, `Style`, and `Color` (that we call `All`) is synthesized by the following command:

```
composer -target=All Base Style Color
```

As we mentioned in Chapter 3, AHEAD provides design rules to validate composition. For further details consult [1].

6.3 AspectJ Implementation of QPL

We now show how QPL is implemented with aspects. The techniques we use are identical to those needed to translate an AHEAD code base to an AspectJ code base in Section 6.5.

Base Feature. This feature is implemented as (38). As we saw in Chapter 4, AspectJ refers to standard Java classes and interfaces as *base code*. We also saw, that AspectJ does not provide a mechanism to modularize multiple classes or aspect files into features [106]. For our implementation of QPL, we use file directories to group the classes and aspects that constitute features.

Style Feature. This feature is implemented with two aspects. The first one uses an introduction to override method `draw` inherited from `Quadrilateral`. The aspect is declared `privileged` to have access to private members of the classes involved and the name is formed with the feature and class names implemented by the aspect. The rationale behind these two decisions is explained shortly.

```
privileged aspect Style_Rectangle {
  void Rectangle.draw() {
    super.draw(); … fill pattern …
  }
}                                                              (44)
```

The second aspect uses an `around` advice on the `execution` of method `draw` on `Trapezoid`. For simplicity, we introduced a new method `style$draw` to `Trapezoid` to hold the body of the method replacement. Thus, any time `draw` is executed, the call is intercepted and redirected to method `style$draw`, effectively replacing the original `draw` method for this class in `Base` feature. The pointcut descriptor of the advice contains a `target` clause to obtain a reference, we call it `obj$Trapezoid`, of the object whose method is executed. This reference is used to access instance members on that object thus

the need for `privileged` aspects in case members are private. In our example, we use this reference to call method `style$draw`.

```
privileged aspect Style_Trapezoid {
 void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid){
       obj$Trapezoid.style$draw();
 }
 void Trapezoid.style$draw(){… dashed lines … }
}                                                              (45)
```

Arguably, this feature could be implemented in a single aspect that combines the contents of aspects (44) and (45). We have seen in Chapter 5 that, in general, aspect composition is hindered by the rules that govern advice precedence. Shortly we describe how we avoided those problems in the implementation of QPL.

Color Feature. This feature is implemented with three aspects. The first one introduces `color` and `setColor` to `Quadrilateral`.

```
privileged aspect Color_Quadrilateral {
  int Quadrilateral.color;
  void Quadrilateral.setColor(int c) {color=c;}
}                                                              (46)
```

The second aspect uses `around` advice on `execution` of method `draw`. It also uses a `target` clause to get the object being called and uses this reference to set the color to `GREEN`. As opposed to the advice in feature `Style`, this aspect uses AspectJ `proceed` statement to continue with the execution of the method. Thus, any time that `draw` is called, the call is intercepted, the color is set, and the execution is resumed.

```
privileged aspect Color_Rectangle {
 void around(Rectangle obj$Rectangle) :
    execution(void Rectangle.draw()) &&
     target(obj$Rectangle){
      obj$Rectangle.setColor(GREEN);
```

```
        proceed(obj$Rectangle);
    }
}                                                                    (47)
```

One way to think about `proceed` in method extensions is as a placeholder of the derived method body we mentioned in (40) because it indicates the execution of the rest of extensions made to this method by other features.

The third aspect follows along the same lines of the previous one only applied to `Trapezoid` instead of `Rectangle`.

```
privileged aspect Color_Trapezoid {
 void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid){
      obj$Trapezoid.setColor(BLUE);
      proceed(obj$Trapezoid);
 }
}                                                                    (48)
```

Product Line Members. AspectJ compiler (weaver) `ajc`, uses the file names of base code and aspects of the features to create product members. Recall from Chapter 4 and Chapter 5 that aspect precedence determines the order aspect introductions and pieces of advice are woven to base code.

In the case of introductions, the one with higher precedence overrides (replaces) those with lower precedence. An introduction cannot override a member already present in base code. This is why class refinement `Trapezoid` in feature `Style` is implemented with an `around` advice.

In the case of pieces of advice, when several of them apply to the same join point, AspectJ follows ordering rules that depend on where the conflicting pieces of advice are defined [11][93]. However, as we demonstrate in Chapter 5, these rules can lead to undefined orders (programmers cannot easily infer the order by looking at the code), circularity errors (compiler cannot not infer a weaving order), and some composition orders cannot be expressed [108].

To prevent problems with advice weaving order, we use only `around` advice and a define an aspect with a `declare precedence` clause to order pieces of advice by features. For instance, the following aspect specifies the weaving order required to obtain the composition of `Base`, `Style`, and `Color` in (40):

```
aspect Ordering {
  declare precedence: Color_*, Style_*;
}                                                    (49)
```

This aspect instructs the compiler to weave first the aspects from the `Style` feature and then those from the `Color` feature, according to the ordering rules that apply to `around` advice in different aspects [11][93]. This is the reason why we followed the convention of including feature names as part of the names of our aspects.

Finally, the `ajc` command for the composition in (40) is (we omit feature directory names for simplicity)[2]:

```
ajc Quadrilateral.java Rectangle.java   Trapezoid.java
  Style_Rectangle.java Style_Trapezoid.java
  Color_Quadrilateral.java   Color_Rectangle.java
Color_Trapezoid.java   Ordering.java                 (50)
```

The order of files in this command is immaterial. Composition validation according to design rules is not part of `ajc`.

6.3.1 Choice of Pointcuts

AspectJ provides a vast array of pointcuts that could be used for feature implementation. We use `execution` pointcuts because their semantics most closely resembles what AHEAD uses for method extension and replacement in QPL. However, there may be cases where other pointcut combinations could be more appropriate.

For example, consider QPL product that contains `Base` and `Color` features. From a design point of view, this program seems completely reasonable and valid; however, its

2. It should be Base/Quadrilateral.java, Base/Rectangle.java, Base/Trapezoid.java, etc.

implementation is not possible using `execution` pointcuts. This is because AspectJ semantics specifies that when a declaring type is used in an `execution` pointcut this captures only join points that match methods declared or overridden in the declaring type [11][12]. In our example, this means that pointcut `execution(void Rectangle.draw())` of feature **`Color`** in (47) matches join points only if `Rectangle` declares or overrides method `draw`, which is not the case in feature `Base` as `Rectangle` inherits the method from `Quadrilateral` (38). This problem can be solved replacing the `execution` pointcut by a combination of `call` and `target` pointcuts.

This example suggests the existence of patterns of method extensions and replacements that could be better expressed with different combinations of pointcuts, new pointcut constructs tailored for specific feature extensions, or semantically modified versions of existing pointcuts [19]. We believe this is an interesting venue for future research.

6.4 Emulating Functional Composition

In Section 5.5.6 we presented the Pair Model of aspect composition and illustrated its implications to common software engineering practices. The implementation in AspectJ of QPL and our case study, the AHEAD tool suite, are implemented using `ajc` which follows the Pair Model. We saw in Chapter 3 that the original AHEAD tool suite implemented in `Jak` follows a functional model similar to that described in Section 5.5.7. In Section 5.8 we analyzed the relationship between these last two models. The question is: Why was it possible to implement the AHEAD tool suite in AspectJ if AHEAD and AspectJ follow different composition models? It turns out that the AspectJ translation of the AHEAD tool suite emulates function composition. In other words, the set of AspectJ language constructs chosen for the translation and the particular usage of precedence does not exhibit any of the problems of the pair model that the Functional Model exposes.

In this section we illustrate how function composition was emulated in QPL, and hence in our case study. Saying that the AspectJ implementation of QPL emulates functional composition means that for the three program members of QPL the Pair Model

yields the same composition expressions as the Functional Model. We start with an intuitive proof that we later formalize using the models of Chapter 5.

In previous chapters we have seen that function composition makes explicit the order of composition and binds the scope to which program extensions are applied. Consider for instance, the program that contains features `Base`, `Style` and `Color` in that order. This program can be denoted as function composition `Color(Style(Base))` because:

- Composition order: Aspect `Ordering` in (49) tells the weaver to first apply to `Base` the pieces of advice and introductions in `Style` followed by those of `Color`.
- Bounded scope: Feature `Style` applies its extensions only to `Base`, and feature `Color` applies its extensions only to its input program `Style(Base)`.

Elaborating more on the second bullet, feature `Style` overrides method `draw` in class `Rectangle` (44), and replaces method `draw` in class `Trapezoid` using advice (45). In the first case, the introduction overrides an inherited method in `Rectangle` of feature `Base`. In the second case, the advice captures join points triggered by the execution of method `draw` of `Trapezoid` also in feature `Base`. Thus feature `Style` applies its extensions only to elements in feature `Base`.

Similarly, feature `Color` applies its extensions only to the program it receives as input `Style(Base)`. Its three refinements ((46),(47), and (48)) add and extend elements present in feature `Base` and thus present in `Style(Base)`. Figure 6.1 illustrates this program. The scoping arrows depict the code that class refinements affect. For example, the two refinements in feature `Style`, affect their corresponding base code classes in feature `Base`. Note that all scoping arrows point upwards, which means that features apply their changes to the programs they receive as input.

It follows from this discussion that programs: a) with feature `Base` and b) with features `Base` and `Style` can also be expressed with functional composition for the same arguments just mentioned.

This findings can be formalized as follows. We start by describing the algebraic terms of the three features of QPL.

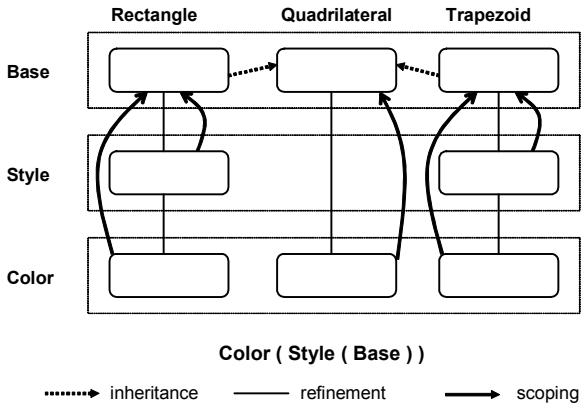


Figure 6.1 Function Composition in QPL

Base Feature. This feature consists of base programs, thus it is represented as:

```
Base = Quadrilateral + Rectangle + Trapezoid                (51)
```

where each term correspond to each of the classes.

Style Feature. This feature contains aspects **S**tyle_**R**ectangle (44) and **S**tyle_**T**rapezoid (45) whose pair representation is respectively defined as:

```
SR = <id, draw>
ST = <sta, staI + style$draw> where
sta is void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid)--> staI ( obj$Trapezoid);
static void Trapezoid.staI(Trapezoid t) {
  t.style$draw();
}
void Trapezoid.style$draw(){… dashed lines … }              (52)
```

Color Feature. This feature contains aspects Color_Quadrilateral (46), Color_Rectangle (47) and Color_Trapezoid (48) whose pair representation is respectively defined as[3]:

```
CQ = <id, color + setColor>
CR = <cra, craI> where
```

3. In this model proceed calls preserve their semantics.

```
cra is void around(Rectangle obj$Rectangle) :
    execution(void Rectangle.draw()) &&
     target(obj$Rectangle) --> craI(Rectangle r);
static void Rectangle.craI(Rectangle r) {
      t.setColor(GREEN); proceed(t);
 }
CT = <cta, ctaI> where
cta is void around(Trapezoid obj$Trapezoid) :
    execution(void Trapezoid.draw()) &&
     target(obj$Trapezoid) --> craI(obj$Trapezoid);
static void Trapezoid.ctaI(Trapezoid t) {
      t.setColor(GREEN); proceed(t);
 }                                                              (53)
```

In QPL the composition order was determined by the precedence clause in aspect `Ordering` (49). Having defined our algebraic terms we now show how the Functional Model of composition can be emulated for the three QPL programs expressed in the Pair Model.

Program with feature `Base`. This follows trivially from the fact that `Base` feature is a constant program.

Program with features `Base` and `Style`. This program is expressed as[4]:

```
ST ◊ SR ◊ Base
  // substitution from (51)and (52)
= <sta, staI + style$draw> ◊ <id,draw> ◊ <id,Base>
  // by pair model composition (33)
= <sta • id • id, staI + style$draw + draw + Base>
  // by pair model weaving rule (36)
= sta • id • id (staI + style$draw + draw + Base)
  // by identity of advice sum of
```

4. Terms `ST` and `SR` can be composed in any other and yield the same result because they affect different base classes.

```
= sta • id (staI + style$draw + draw + Base)
// by partial distrib. of weaving over introduction sum
= sta (id(staI) + id(style$draw) + id(draw + Base))
// by definition of id
= sta (staI + style$draw + id(draw + Base))
// by functional model of composition (37)
= ST(SR(Base))
```

Program with features `Base`, `Style`, and Color. This program is expressed as follows[5]:

```
CT ◊ CR ◊ CQ ◊ ST ◊ SR ◊ Base
// by substitution from (51), (52), and (52)
= <cta,ctaI> ◊  <cra,craI> ◊ <id,color+setColor> ◊
  <sta, staI + style$draw> ◊  <id,draw> ◊ <id,Base>
// by pair model composition (33)
= <cta •cra • id •sta • id • id,
  ctaI+ craI+ color+setColor+ staI+style$draw + draw +Base>
// by pair model weaving rule (36)
= cta •cra • id •sta • id • id (ctaI + craI + color+ setColor
 + staI + style$draw + draw + Base)
// by partial distributivity of weaving and definition of id
= cta •cra • id •sta • id (ctaI + craI + color+ setColor +
 staI + style$draw + id (draw + Base))
// by definition of id
= cta •cra • id •sta (ctaI + craI + color+ setColor +
 staI + style$draw + id (draw + Base))
// by partial distributivity of weaving[6]
= cta •cra • id [ sta (ctaI + craI + color+ setColor) +
                  sta(staI + style$draw + id (draw + Base))]
```

5. Additionally terms CT, CR, and CQ can appear in any other and yield the same result.
6. Brackets are for visual aid only, they have the same meaning as parenthesis.

```
// by distributivity and fixed point of weaving
= cta •cra • id [ ctaI + craI + color+ setColor +
                sta(staI + style$draw + id (draw + Base))]
// by partial distributivy and definition of id
= cta •cra [ ctaI + craI + id (color+setColor +
                  sta(staI + style$draw + id(draw+Base)))]
// by partial distributivity and fixed point
= cta(ctaI + cra(craI + id (color+setColor +
                  sta(staI + style$draw + id(draw+Base)))))
// by functional model of composition (37)
= CT ( CR ( CQ ( ST ( SR ( Base )))))
```

6.5 AHEAD Translation Case Study

Section 6.2 and Section 6.3 describe how AHEAD and AspectJ implement QPL, and illustrate the mapping between both approaches that was followed for the translation of the AHEAD tool suite case study.

We translated into aspects the code base of the five key tools of AHEAD: a) *mixin* performs mixin composition on features, b) *jampack* composes features by collapsing their refinement chains, c) *unmixin* propagates changes from composed files back to their constituent features, d) *jak2java* translates Jak programs into Java, and e) *mmatrix* supports AHEAD feature browser.

6.5.1 Mapping Jak to AspectJ

The mapping between Jak and AspectJ constructs was implemented in a translator called *jak2aj*. It is summarized in Figure 6.2. Though not shown in the figure, the translation of interfaces is similar to that of classes.

There are four special cases in this mapping:

	Jak	AspectJ
Standard Class	layer L; class C { … }	class C { …}
New field	layer L; refines class C { mods type f; }	privileged aspect L_C { mods type C.f; }
New method	layer L; refines class C { mods type f(args) {...} }	privileged aspect L_C { mods type C.f(args) {...} }

Method override	layer L;
	refines class C {
	overrides mods type f(args) { ... Super.f(args); ... }
	}
		privileged aspect L_C {
		mods type C.f(args) { ... super.f(args); ... }
		}

Method replacement	layer L;
	refines class C { new mods type f() { ... } }
		privileged aspect L_C {
		type around(C obj$C) : execution(mods type C.f())
		&& target(obj$C) { return obj$C.L$f( ); }
		type C.L$f() { … }
		}

Method extension	layer L;
	refines class C { mods type f() { ...Super.f(); … } }
		privileged aspect L_C {
		type around(C obj$C) : execution(mods type C.f())
		&& target(obj$C) { …proceed(obj$C); … }
		}

Figure 6.2 jak2aj Translation Summary

- Translation of static methods does not have `target` clauses because their execution is associated to a class not to an object.
- Methods with arguments `and` (`&&`) an extra `args` pointcut for the method parameters which are bound to the around advice parameters.
- AspectJ does not permit introduction of protected members [12]. We translated those as public members.
- Interfaces in `implements` clauses of class refinements are mapped to `declare parent` clauses in their corresponding aspect.

AspectJ has an asymmetrical approach to overriding [11][12]; precedence can override introductions but not base code members. Because of this, we had to distinguish between method override and method replacement.

6.5.2 Results

The five tools studied are built from combinations of 48 different features. The code generated for all of them is slightly more than 205K+ LOC (Figure 6.3a). The 48 features surveyed are implemented in 524 standard Java files (base code) and 503 aspect files. In terms of LOC, Java code is 38K+ and AspectJ code 18K+. Thus, we found a 68%-32%

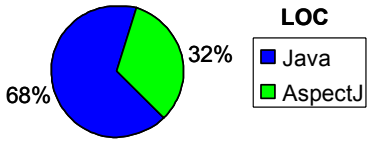
ratio between Java code and AspectJ code as illustrated in Figure 6.3b. The Java code has 1006 fields, 40 constructors and 2200+ methods. AspectJ code introduces 58 new fields and 610 new methods, with 164 method overrides, 8 method replacements, and 8 method extensions (Figure 6.3c). These numbers and their corresponding LOC indicate that AHEAD tool suite relies heavily on introductions, and only uses a tiny number of pieces of advice. Of the 56700+ LOC in the 48 features of the synthesized tools, only 119 lines (.2% — 2 tenths of one percent) is due to method extension, and 440 lines (.7% — seven tenths of one percent) is due to method replacement.

A strength of AOP is without doubt the sophisticated mechanisms to specify complex pointcuts yet in the case of AHEAD tool suite they were not fully exploited. This is not surprising, as AHEAD itself offers very limited advising capabilities (method and constructor extensions). This explains why less than 1% of the synthesized code is due to advice. An open research question is: if AHEAD had more powerful forms of pointcuts and advice, how much larger would this percentage be?

Tool	Features	LOC
mixin	16	40402
jampack	20	44234
unmixin	11	38798
jak2java	17	43907
mmatrix	12	39712
	Total	207053

(a)

	Java	AspectJ
Num Files	524	503
LOC	38300	18427



(b)

	Java	AspectJ – LOC
Field	1006	58 – 58
Const	40	0 – 0
Method	2238	610 – 9295
Override		164 – 1574
Replace		8 – 440
Extension		8 – 119

(c)

Figure 6.3 AHEAD Case Study Summary

A bound on this number is the recognition that features in product lines generally implement collaborations [7][147], which in the AOP literature correspond to *heterogeneous crosscuts*. Such crosscuts deal with field and method introductions and advice whose pointcuts qualify a single join point. AspectJ excels in realizing *homogeneous crosscuts* (advice whose pointcuts qualify multiple join points). The important role of heterogeneous crosscuts is not surprising: large programs are not synthesized by adding the same piece of code in different places, but rather, adding different pieces of code in different places.

6.6 Related Work

The use of AOP as an implementation technology for product lines have been analyzed and evaluated in several case studies: embedded systems [36][100], graph algorithms [104], extensibility problem [106], mobile phones applications [5][4], middleware software [50][163], and e-commerce [124]. We elaborate on those closest to our work.

Anastasopoulus and Muthig propose criteria to evaluate AOP as a product line implementation technology [5]. Their criteria encompasses the main activities of both framework and application engineering. They regard AOP as a program transformation technique. From that perspective, their evaluation on *Reuse Over Time* concludes that aspect reuse is hindered because it is hard to predict the effects of AOP transformations. The foundations of our function composition model is an algebra that also regards aspects as transformations [108]. This perspective allowed us to analyze AspectJ composition and propose an alternative model with simpler and more predictable composition semantics. Another evaluation criterion is *Variation Type* where authors mention precedence as a mechanism to address variation order. In this paper we showed that in some cases precedence clauses are not enough to express some variation orders.

Alves et al. propose a methodology that combines reactive and extractive approaches to product line development [4]. It is an iterative process that starts by identifying concerns in a set of related products, builds a concern graph [141], and extracts the

commonality and variability present in the graph to build a product line design. At each iteration the product line is reactively adapted through code transformations based on a set of template-oriented AOP refactorings. This contrasts with our work as we use aspects as building blocks of an existing product line. However, we believe that a function composition model can also increase reuse in the aspects extracted following their methodology.

Coyler and Clement implemented members a middleware product line using aspects [50]. They developed three features that are typical AOP applications: tracing, monitoring and failure data capture. They also refactored into a feature the support for EJB in a server application. Their implementation of this latter feature shows a 3-1 ratio between the number of introductions and pieces of advices. More surprisingly, the pieces of advice of this feature capture only a single join point as we did in our case study.

A similar study was performed by Zhang and Jacobsen [163]. They refactored aspects from a CORBA implementation using an iterative process they called *Horizontal Decomposition (HD)*. They achieved a 40% reduction on code size and good performance improvement. Liu and Batory have proposed an algebraic theory of feature composition and decomposition that generalizes and provides a mathematical foundation in which to understand HD [99]. This theory also relies on function composition.

Chapter 7

Conclusions and Future Work

This dissertation proposed features as modularity units for abstraction and synthesis of programs. We developed algebraic models and tools to support feature modules, assess their applicability by implementing a non-trivial case study, and evaluated support for features in emerging modularization technologies. In this chapter we review the core results and contributions of our work, and present venues for future research.

7.1 Results and Contributions

Software systems are commonly described and designed by features, and the functionality they provide. Unfortunately conventional software modularization paradigms fail to preserve feature abstraction down to the implementation level because of two problems. First, a typical feature implementation is spread over several conventional modules. Second, features are usually more than source code artifacts as they can modularize many different program representations such as makefiles, documentation, or performance models.

The lack of feature modularity causes software developers to lower their abstractions from features to those provided by the underlying implementation languages. In other words, features are broken into many fragments which (at best case) become modular units in supporting languages or (most commonly) form new modular units with fragments from other features. Thus the conceptual gap between feature abstractions and their modularization hinders program understanding and product line development.

We proposed features, increments in program functionality, as an evolutionary step in program abstraction and modularity. We explored the foundations of Feature Oriented Programming, a software development paradigm whose cornerstone is feature modularity, and explained its algebraic model, underlying principles, and tool support.

A key contribution of our work was Origami [27], an architectural model to simplify program specification. Specifying a program as a composition of feature modules can quickly become unwieldy when the number of feature combinations explodes as the number of modules increases. Origami addresses this problem by constructing n-dimensional models, depicted as n-dimensional matrices. Each dimension is formed by an AHEAD model, and feature modules are arranged in the matrix at the n-dimension intersection of the functionality they implement. Origami specifies a composition for each dimension with a dimensional equation. In the general case, a model with `k` dimensions and `n` features per dimension, this significantly reduces the length of specification from $O(n^k)$ to $O(nk)$. This reduction is crucial because it allows AHEAD to seamlessly scale program synthesis to product lines of non-trivial size and complexity.

To make an assessment of feature modularity support in emerging modularization technologies, we developed an example product line – Expressions Product Line (EPL), yet another example of Origami – and proposed basic properties that feature modules that implement this problem should provide. We evaluated several technologies and found that none of them fully satisfy all the proposed properties. We related the properties to our algebraic models, thus providing a foundation for an implementation-independent model on which to compare and contrast modularization technologies.

AspectJ is the flagship language of Aspect Oriented Programming, a popular and emerging modularization technology. We developed an algebraic model for a core of this language which helped us highlight two problems with its defacto composition model: unbounded quantification, and aspect composition via precedence clauses. We devised an alternative model, which regards aspect composition as function composition, to address these problems. We believe this model in conjunction with the model of FOP constitutes a foundation of a mathematically-founded model of feature modularity.

We performed an empirical assessment of the functional model of aspects. We implemented the core tools of ATS (200K+ LOC) in AspectJ to assess the validity and relevance of our model. This required the construction of a translator from Jak to AspectJ whose mapping emulated functional composition by preventing the problems identified with the defacto AspectJ model. Statistics were collected that shed light on the impact different constructs have on feature modules and on the overall structure of the product line.

7.2 Future Work

Despite its success in implementing systems of non-trivial scale, we believe Feature Oriented Programming is still in its infancy and has a promising potential. Overall it requires a more cohesive and comprehensive tool support, coupled with a thorough implementation and assessment of case studies from a wide spectrum of problem domains. To contribute towards those goals, our work opens several concrete venues for future research:

- Generalization of AHEAD model to include aspects and the corresponding development of supporting tools. We emulated functional composition for the implementation of five core tools of ATS by avoiding the problems highlighted in the defacto model of AspectJ. This case study helped us glimpse at the potential effect of functional composition in program development with aspects. However, to fully gauge and exploit the impact that aspects may have on product line synthesis we need to generalize AHEAD composition model and develop a set of supporting tools. For the implementation of these tools, we may rely on parts of the infrastructure provided by `abc` [10][17], an extensible AspectJ compiler.
- Analysis of aspect constructs impact on product line implementation. The statistics we gathered for the implementation of ATS showed that its features are predominantly collaborations, heterogeneous crosscuts in AOP terms, which can be elegantly implemented with OO technology similar to mixins. On the other hand, AspectJ excels at realizing homogeneous crosscuts; however, our case study did not require crosscuts of this type. Based on the experience of other researchers and ours

[7], we conjecture that the predominant use of aspects in product lines is to implement heterogeneous crosscuts rather than homogeneous crosscuts. To validate this conjecture it is necessary to develop a set of metrics to quantify how aspect language constructs are used and how they affect the structure of feature code to implement heterogeneous or homogeneous crosscuts. These metrics would then be applied to a pool of AspectJ-based product lines and then our conjecture could be evaluated.

- Compiler and type theory support for FOP. The current implementation of AHEAD is based on sophisticated program preprocessors. This technology presents limitations on the kinds of program analysis that can perform. We are interested in statically checking that feature modules and their composition is well typed. Thus a type theory and appropriate compiler support are needed.
- Broadening the scope of FOP for product line development. In Chapter 2 we described the two main processes of product line methodologies, Domain and Application Engineering, and sketched their basic activities. Throughout the dissertation, we have seen that FOP touches on the activities of design and configuration, but fundamentally addresses implementation and generation. We believe FOP can extend its scope to other activities. For instance, work on UML support for product line design [68], and implementation of use cases with AspectJ [80][82] open the possibility of integrating FOP into mainstream design practices using models such as UML.

Bibliography

[1] AHEAD Tool Suite (ATS). http://www.cs.utexas.edu/users/schwartz

[2] Aldrich, J.: Open Modules: Modular Reasoning about Advice. ECOOP (2005)

[3] Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM TOSEM, Vol. 6, No. 3 (1997)

[4] Alves, V., Matos, P., Cole, L., Borba P., Ramalho G.: Extracting and Evolving Game Product Lines. SPLC (2005)

[5] Anastasopoulus, M., Muthig, D.: An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. ICSR (2004)

[6] Anastasopoulus, M., Gacek, C.: Implementing Product Line Variabilities. SSR (2001)

[7] Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. ICSE (2006)

[8] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. Transactions on AOSD, Vol. 1 (2006)

[9] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a Common Component for High-Performance Scientific Computing. HPDC (1999)

[10] Aspect Bench Compiler. www.aspectbench.org

[11] AspectJ Manual. http://eclipse.org/aspectj/doc/progguide/language.html

[12] AspectJ Version 1.2. http://eclipse.org/aspectj

[13] AspectJ Developers mailing list. http://dev.eclipse.org/mhonarc/lists/aspectj-dev/maillist.html. Thread on execution order. October 5, 2005.

[14] AOSD Europe Network of Excellence Workshop. ECOOP (2005)

[15] AOSD Europe Network of Excellence. Survey of Aspect-Oriented Languages and Execution Models. http://www.aosd-europe.net

[16] Atkinson, C.: Component-based Product Line Engineering with UML. Addison-Wesley (2001)

[17] Avgustinov, P. et al.: abc: An Extensible AspectJ Compiler. AOSD (2005)

[18] Avgustinov, P. et al.: Optimizing AspectJ. PLDI (2005)

[19] Barzilay, O., Tyszberowicz, S., Feldman, Y.A., Yehudai, A.: Call and Execution Semantics in AspectJ. FOAL Workshop AOSD (2004)

[20] Bassett, P.G.: Framing Software Reuse. Lessons from the Real World. Yourdon Press Computing Series, Prentice Hall (1997)

[21] Batory, D: Feature Models, Grammars, and Propositional Formulas. SPLC (2005)

[22] Batory, D.: Feature Oriented Programming. Class Notes. Spring 2006. Department of Computer Sciences. The University of Texas at Austin.

[23] Batory. D., Cardone, R., Smaragdakis, Y.: Object-Oriented Frameworks and Product-Lines. SPLC (2000)

[24] Batory, D., Chen, G., Robertson, E., Wang, T.: Design Wizards and Visual Programming Environments for GenVoca Generators. IEEE TSE, May (2000)

[25] Batory, D., Geraci, B.J,: Composition Validation and Subjectivity in GenVoca Generators. IEEE TSE, February (1997) 67-82

[26] Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. ACM TOSEM, April (2002)

[27] Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating Product-Lines of Product-Families. ASE (2002)

[28] Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multidimensional Separation of Concerns. ACM SIGSOFT, September (2003)

[29] Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components, ACM TOSEM, October (1992)

[30] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE, June (2004)

[31] Baxter, I.D.: Design Maintenance Systems. CACM, Vol. 55, No. 4 (1992) 73-89

[32] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: PuLSE: A Methodology to Develop Software Product Lines. SSR (1999)

[33] Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A Minimal Module Model Supporting Local Rebinding. Joint Modular Languages Conferences JMLC (2003)

[34] Bergel, A., Stephane Ducasse, Oscar Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. OOPLA (2005)

[35] Beuche, D.:Composition and Construction of Embedded Software Families. Ph.D. Otto-von-Guericke-Universität Magdeburg (2003)

[36] Beuche, D., Spinczyk, O.: Aspect-Oriented Product Line Development in Constrained Environments. Workshop on Reuse in Constrained Environments OOPSLA (2003)

[37] Bezivin, J.: From Object Composition to Model Transformation with the MDA. TOOLS'USA, August (2001)

[38] Bracha, G., Cook, W.: Mixin-based inheritance. OOPSLA (1990)

[39] Carnegie Mellon University, Software Engineering Institute, Framework for Software Product Line Practice. http://www.sei.cmu.edu/productlines/framework.html

[40] Chiba, S.: Program Transformation with Reflective and Aspect-Oriented Programming. Generative and Transformational Techniques in Software Engineering (2005)

[41] Clarke, S., Walker, R.: Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J. Technical Report UBC-CS-TR-2001-05, University of British Columbia, Canada (2001)

[42] Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley (2002)

[43] Clifton, C.: A Design Discipline and Language Features for Modular Reasoning in Aspect-Oriented Programs. Ph.D. Dissertation, Dept. Computer Science, Iowa State (2005)

[44] Clifton, C., Leavens, G.T., Millstein, T., Chambers, G.: MultiJava: Modular Open classes and Symmetric Multiple Dispatch for Java. OOPSLA (2000)

[45] Clifton, C., Leavens, G.T.: Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. SPLAT (2003)

[46] Clifton, C., Millstein, T., Leavens, G.T., Chambers, G.: MultiJava: Design Rationale, Compiler Implementation, and User Experience. TR #04-01, Iowa State University (2004)

[47] Concern Manipulation Environment (CME). http://www.eclipse.org/cme/

[48] Cook, W.R.: Object-Oriented Programming versus Abstract Data Types. Workshop on FOOL, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178

[49] Coyler, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. TRCOMP-001-2004, Computing Department, Lancaster University, UK (2004)

[50] Coyler, A., Clement, A.: Large-scale AOSD for Middleare. AOSD (2004)

[51] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[52] Diaz, M., Rubio, B., Soler, E., Troya, J.M.: SBASCO: Skeleton-based Scientific Components. EUROMICRO-PDP (2004)

[53] Dijkstra, E.W.: The Structure of the 'THE'-Multiprogramming System. CACM, May (1968)

[54] Dijkstra, E.W.: On the role of scientific thought. EWD-447. Computer Sciences Department, The University of Texas at Austin.

[55] Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)

[56] Dounce, R., Le Botlan, D.: Towards a Taxonomy of AOP Semantics. AOSD-Europe. Technical Report, July (2005)

[57] Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. AOSD (2004)

[58] Driver, C.: Evaluation of Aspect-Oriented Software Development for Distributed Systems. Master's Thesis, University of Dublin, Ireland, September (2002)

[59] Early Aspects website. http://www.early-aspects.net

[60] Ernst, E.: Family Polymorphism. ECOOP (2001)

[61] Ernst, E., Ostermann, K., Cook, W.: A Virtual Class Calculus. POPL (2006)

[62] Eriksson, M., Börstler, J., Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realization. SPLC (2005)

[63] Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)

[64] Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. PLDI (1998)

[65] Findler, R.B., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. ICFP (1998) 94-104

[66] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)

[67] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Second Edition. Prentice Hall (2003)

[68] Gomaa, H.: Designing Software Product Lines with UML. From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)

[69] Gorlick, M.M., Razouk, R.R.: Using Weaves for Software Construction and Analysis. ICSE (1991)

[70] Gray, J., et al.: A Technique for Constructing Aspect Weavers Using a Program Transformation Engine. AOSD (2004)

[71] Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with RSEB. International Conference on Software Reuse (1998)

[72] Gybels, K, Brichau, J.: Arranging Language Features for More Robust Pattern-based crosscuts. AOSD (2003)

[73] Gybels, K., Ostermann, and K. Discussions at SPLAT (2005)

[74] Herrmann, S.: Object TeamsL Improving Modularity for Crosscuting Collaborations. NODe (2002)

[75] Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. AOSD (2004)

[76] Holmes, C., Evans, A.: A Review of Frame Technology, Technical Report YCS-2003-369, York University, UK (2003)

[77] Hutchins, D.: Making Inheritance Scale: Towards a Theory of Deep Mixin Composition. Univ. of Edinburgh (2005)

[78] Ichisugi, Y., Tanaka, A.: Difference-Based Modules: A Class-Independet Module Mechanism. ECOOP (2002)

[79] Ichisugi, Y.: MixJuice, http://staff.aist.go.jp/y-ichisugi/mj

[80] Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Bussiness Success. Addison-Wesley (1997)

[81] Jacobson, I.: Use cases and Aspects — Working Seemlessly Together. Journal of Object Technology. vol. 2, no. 4, July-August (2003)

[82] Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)

[83] Jagadeesan, R., Jeffrey, A., Riely, J.: A Typed Calculus of Aspect Oriented Programs. Submitted for publication.

[84] Javadoc — The Java API Documentation Generator. Sun Microsystems, http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html

[85] Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-21, Carnegie Mellon Univ., Pittsburgh, PA, Nov. (1990)

[86] Kang, K., Lee, J., Donohoe, P. Feature-Oriented Product Line Engineering. IEEE Software, Vol. 19, No.4 (2002)

[87] Kiczales, G., Hilsdale, E., Hugunin, J., Kirsten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. ECOOP (2001)

[88] Kiczales, G.: Personal email communication (2003)

[89] Kiczales, G., Mezini M.: Aspect-Oriented Programming and Modular Reasoning. ICSE (2005)

[90] Kniesel, G., Costanza, P., Austermann, M.: JMangler - A Framework for Load-Time Transformation of Java Class Files. SCAM (2001)

[91] Kreitz, C.: Automated Deduction - A Basis For Applications. Chapter III.2.5 Program Synthesis. Kluwer (1998)

[92] Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying Aspect Advice Modularity. FSE (2004)

[93] Laddad, R.: AspectJ in Action. Practical Aspect-Oriented Programming. Manning (2003)

[94] Lämmel, R.: Declarative Aspect-Oriented Programming. PEPM (1999)

[95] Lämmel, R., Saraiva, J., Visser, J. (Eds): Generative and Transformational Techniques in Software Engineering (2005)

[96] Lehrmann Madsen, O., Moller-Pedersen, B.: Virtual Classes: A Powerful Mechanism in Object Oriented Programming. OOPSLA (1989)

[97] Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal, 46(5):542--565, September (2003)

[98] Liu, J., Batory, D.: Automatic Remodularization and Optimized Synthesis of Product-Families. GPCE (2004)

[99] Liu, J., Batory, D.: Feature Oriented Refactoring of Legacy Applications. ICSE (2006)

[100] Lohmann, D., Spinczyk, O., Schröder-Preikschat, W.: On the Configuration of Non-Functional properties in Operating System Product Lines. ACP4IS Workshop AOSD (2005)

[101] Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting Product Line Evolution with Framed Aspects. ACP4IS Workshop, AOSD (2004)

[102] Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. ICSR (2004)

[103] Lopez-Herrejon, R.E., Batory, D.: Jedi: A Documentation Generator for Extensible Domain-Specific Languages. Young Researchers Workshop, Seventh International Conference on Software Reuse (2002)

[104] Lopez-Herrejon, R.E., Batory, D.: Using AspectJ to Implement Product-Lines: A Case Study. Technical Report. Department of Computer Sciences. University of Texas at Austin. TR-02-45. September 2002.

[105] Lopez-Herrejon, R.E., Batory, D.: Improving Incremental Development in AspectJ by Bounding Quantification. SPLAT Workshop at AOSD (2005)

[106] Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. ECOOP (2005)

[107] Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. ECOOP (2005) Extended Report

[108] Lopez-Herrejon, R.E., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. PEPM (2006)

[109] Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. ESEC (1995)

[110] Mahmood, N., Deng, G., Browne, J.C.: Compositional Development of Parallel Programs. LCPC (2004)

[111] Mantilassi, M., Niemelä, E., Dobrica, L.: Quality-Driven Architecture Design and Quality Analysis. ESPOO (2002)

[112] Masuhara, H., Kiczales, G.: Modeling Crosscuting Aspect-Oriented Mechanisms. ECOOP (2003)

[113] McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New age components for old-fashioned Java. OOPSLA (2001)

[114] McDirmid, S., Hsieh, W.C.: Aspect-Oriented Programming with Jiazzi. AOSD (2003)

[115] McDirmid, S.: The Jiazzi Manual (2002)

[116] McEachen, M., Alexander, R.T.: Distributing Classes wth Woven Concerns - An Exploration of Potential Fault Scenarios. AOSD (2005)

[117] Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE TOSEM, Volume 26. No.1 January (2000)

[118] Meyer, B.: Object-Oriented Software Constructor. Second Edition. Prentice Hall, 1997.

[119] Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. AOSD (2003)

[120] Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. SIGSOFT04/ FSE-12 (2004)

[121] Misra, J.: A Discipline of Multiprogramming. Springer-Verlag (2001)

[122] Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press (2003)

[123] Murphy, G., Lai, A., Walker, R.J., Robillard, M.P.: Separating Features in Source Code: An Exploratory Study. ICSE (2001)

[124] Nyßen, A., Tyszberowicz, S., Weiler, T.: Are Aspects Useful for Managing Variability in Software Product Lines? A Case Study. Aspects and Product Lines Workshop SPLC (2005)

[125] Object Management Group. http://www.omg.org/

[126] Odersky, M., et al.: An Overview of the Scala Programming Language. September (2004), http://scala.epfl.ch

[127] Odersky, M., et al.: The Scala Language Specification. September (2004), http://scala.epfl.ch

[128] Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. ECOOP (2003)

[129] Ongkingco, N.: Adding Open Modules to AspectJ. AOSD (2006)

[130] Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In Software Architectures and Component Technology, Kluwer (2002)

[131] Ostermann, K.: Dynamically Composable Collaborations with Delegation Layers. ECOOP (2002)

[132] Ovlinger, J.: Combining Aspects and Modules. PhD Dissertation, College of Computer and Information Science, Northeastern University (2004)

[133] Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM, December (1972)

[134] Parnas, D.: On the Design and Development of Program Families. Transactions on Software Engineering. Vol SE-2, March (1976)

[135] Partsch, H., Steinbrüggen, R.: Program Transformation Systems. ACM Computing Surveys, September (1983)

[136] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer (2005)

[137] Rajan, H., Sullivan, K.J.: Classpects: Unifying Aspect- and Object-Oriented Programming. ICSE (2005)

[138] Reynolds, J.C.: User-defined types and procedural data as complementary approaches to data abstraction. Theoretical Aspects of Object-Oriented Programming, MIT Press, (1994)

[139] Rinard, M., Salcianu, A., Bugrara, S.: A Classification System and Analysis for Aspect-Oriented Programs. FSE (2004)

[140] Rho, T., Kniesel, G.: LogicAJ - A Uniformly Generic Aspect Language. Submitted for publication.

[141] Robillard, M., Murphy, G.: Concern graphs: Finding and describing concerns using structural program dependencies. ICSE (2002)

[142] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. ECOOP (2003)

[143] Schinz, M.: A Scala tutorial for Java programmers. September (2004), http://scala.epfl.ch

[144] P. Selinger, et al.: Access Path Selection in a Relational Database System. ACM SIGMOD (1979) 23-34

[145] Semantic Designs. http://www.semdesigns.com

[146] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)

[147] Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM April (2002)

[148] Software Engineering Institute. Predictable Assembly from Certified Components. http://www.sei.cmu.edu/pacc

[149] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, Addison-Wesley (2002)

[150] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE (1999) 107-119

[151] Tarr, P., Ossher, H.: Hyper/J User and Installation Manual. IBM Corporation (2001)

[152] Tokuda, L., Batory, D.: Evolving Object-Oriented Designs with Refactorings. Journal of.ASE 8 (2001)

[153] Torgersen, M.: The Expresion Problem Revisited. Four new solutions using generics. ECOOP (2004)

[154] Vanneschi, M.: The Programming Model of ASSIST, an environment for Parallel and Distributed Portable Applications. Elsevier Parallel Computing 28 (2002)

[155] Wadler, P.: The expression problem. Posted on the Java Genericity mailing list (1998)

[156] Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. ICFP (2003)

[157] Wand, M., Kiczales, G., Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming. TOPLAS (2004)

[158] Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering. A Family-Based Software Development Process. Addison Wesley (1999)

[159] Wirth, N.: Program Development by Stepwise Refinement. CACM 14 #4, (1971) 221-227

[160] Withey, J.: Investment Analysis of Software Assets for Product Lines. Technical Report, CMU/SEI-96-TR-010, SEI, Carnegie Mellon University, (1996)

[161] Xin, B., McDirmid, S., Eide, E., Hsieh, W.C.: A comparison of Jiazzi and AspectJ. Technical Report TR UUCS-04-001, University of Utah (2004)

[162] Zave, P.: FAQ Sheet on Feature Interaction. http://www.research.att.com/~pamela/faq.html

[163] Zhang, C., Jacobsen, H.: Resolving Feature Convolution in Middleware Systems. OOPSLA (2004)

[164] Zhang, H., Jarzabek, S., Swe, S. M.: XVCL Approach to Separating Concerns in Product Family Assets. International Conference on Generative and Component-Based Software Engineering. Lecture Notes in Computer Science, Vol. 2186, (2001) 36-47

[165] Zenger, M., Odersky, M.: Independently Extensible Solutions to the Expression Problem. Technical Report TR IC/2004/33, EPFL Switzerland (2004)

Vita

Roberto Erick Lopez Herrejon was born in Morelia, Michoacan, Mexico on February 24, 1971. He did primary, secondary and high school studies at Instituto Valladolid in Morelia. He discovered the existence of computers by chance when at age 13 he got a Sinclair Z80 as a birthday gift from his mother. It was love at first sight. Two years later, while still in secondary school, he earned a technical degree in Computer Programming from a technical institute in his hometown. Shortly after, he sold his first inventory system to a local handcraft goods retailer. A year later he earned another technical degree as Systems Analyst. In Fall 1989 he entered Tecnologico de Monterrey (ITESM) Campus Queretaro to purse a B.Eng. in Computer Systems which he earned in Fall 1993. After working for some time in an academic environment and some soul searching he decided to purse graduate studies. He studied a Masters Degree in Computer Sciences at IIMAS-UNAM from Fall 1996 - Summer 1998. During that time he was lucky to meet many friends and most importantly his now wife Lupita. He received a Fulbright scholarship to purse studies in Austin. After many happy and challenging years in Austin he got a Career Development Fellowship at the University of Oxford in England where he moved to in September 2005. Besides developing his career towards an academic profession he claims he will use his stay in Britain to learn how to punt, row, and play cricket. He believes that the best is always yet to come and hopes nobody reads this vita and gets a wrong impression of him.

Permanent Address: Circuito Campestre 77, Club Campestre Morelia
Morelia, Michoacan, Mexico CP 58260

This dissertation was typed by the author.