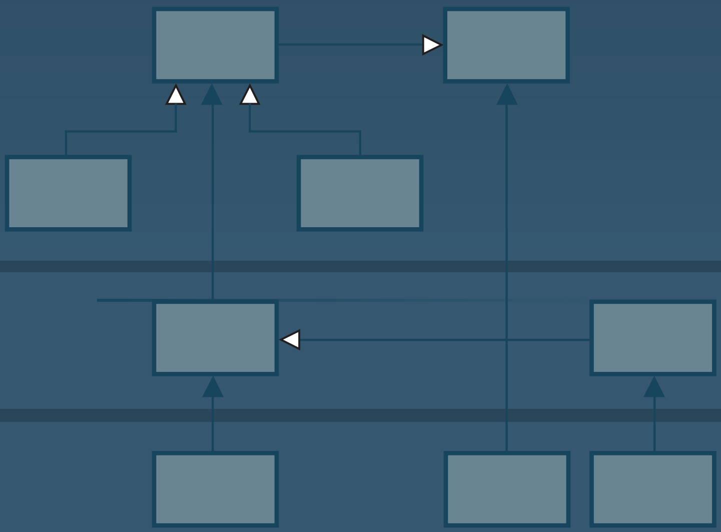
Sven Apel · Don Batory
Christian Kästner · Gunter Saake

Feature-Oriented Software Product Lines

Concepts and Implementation



Springer

Feature-Oriented Software Product Lines

Sven Apel · Don Batory
Christian Kästner · Gunter Saake

Feature-Oriented Software Product Lines

Concepts and Implementation

Springer

Sven Apel
University of Passau
Passau
Germany

Christian Kästner
Carnegie Mellon University
Pittsburgh, PA
USA

Don Batory
The University of Texas
Austin, TX
USA

Gunter Saake
Otto von Guericke University
Magdeburg
Germany

ISBN 978-3-642-37520-0 ISBN 978-3-642-37521-7 (eBook)
DOI 10.1007/978-3-642-37521-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013935777

ACM Computing Classification (1998): D.2, K.6

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

Features are a fundamental notion in modern software engineering. Defined once by Pamela Zave as "incremental units of functionality", they are central to how software is developed now. Much of today's software is developed using scenario-driven approaches. In essence, scenarios, also known as use cases or user stories, are specifications of features.

Feature-oriented software development shines in the context of software product lines. Virtually any successful software faces the need to cater different feature combinations to different customer segments. Product line engineering accelerates product development by leveraging the commonalities among the product line members, while managing the differences, also known as variabilities, among them. Features play a key role in modeling commonalities and variabilities and in managing the development of product lines. Major organizations, including General Motors and Danfoss, use feature-oriented approaches to successfully develop complex software-intensive product lines.

The message of this book is that much of the tremendous power of features is yet to be unlocked by *making features explicit throughout the entire systems and software lifecycle*. The explicit treatment of features in requirements, architecture, implementation, and verification and validation can greatly improve the management of software. Features are abstractions that can be made understandable to all stakeholders, both technical and nontechnical, enabling effective communication among the stakeholders and planning, implementation, and evolution of complex software product lines. Many of the ideas and tools presented in this book are applicable not only to traditional software product lines, but also to a wide range of variability-intensive systems, including highly-configurable applications, computing platforms, and software ecosystems.

The book provides a systematic introduction to feature-oriented software product lines, and leads the reader to more advanced topics in its second half. The authors distill the concepts and principles underlying the field with remarkable clarity, providing a much-needed foundation for the field. They also illustrate these concepts and principles using concrete examples, showcasing languages, tools, and systems from both industrial practice and latest research. The advanced part of the book covers recent research results, many of which the authors have helped to advance. The reader can also enhance his or her learning experience by completing the provided exercises. The book will make an excellent upper-year undergraduate or introductory graduate text; but also practitioners will find it invaluable to enhance their software engineering toolbox with the powerful concepts and techniques of feature-oriented software product lines.

There is no better team than these four authors to write about feature-oriented software product lines. The authors have made fundamental scientific and engineering contributions to the field. Don has pioneered feature-oriented composition of software with his work on GENESIS, an extensible database management system, in late 1980s, and generalized the concepts and principles underlying it in early 1990s. He has continued on this path, advancing the theory and designing languages and tools, but also inspiring generations of researchers to join the effort. I started working in the field after attending Don's tutorial on "Software Systems Generators, Architecture, and Reuse" at the International Conference of Software Reuse in 1996. Shortly after, I shared my excitement for Don's ideas on software generation with Ulrich Eisenecker, which led to our work on automating component assembly based on feature models and, eventually, the book on Generative Programming in 2000. The subsequent decade has seen tremendous progress. New generations of young researchers have worked on techniques for feature-oriented modularization, variability-aware analyses, and empirical studies of systems with variability. Sven and Christian, enjoying the creative and fertile environment of Gunter's research group in Magdeburg and inspired by Don's work, have played leading roles in this progress. Their research results on feature-oriented software product lines have reached wide audiences at major software engineering conferences, such as the International Conference on Software Engineering and the Conference on Foundations of Software Engineering. Today, the four authors are central figures in the growing, vibrant community, known as Feature-Oriented Software Development (FOSD).

The notion of features has already profoundly affected how software is engineered, and this is just the beginning. Features can substantially improve the communication among all stakeholders and will likely lead to new, more effective ways to modularize and develop software. Despite the tremendous progress so far, much potential and many more discoveries lie ahead. Future work topics include finding most effective ways to exploit features in software modularization, creating techniques for re-engineering legacy towards feature orientation and evolving feature-oriented software, and also supporting features more pervasively in tools and infrastructures, such as in configuration management. Years to come will bring new, unexplored topics, which none of us can possibly foresee.

It is a great joy to see the new generations of brilliant young researchers joining the thriving FOSD community. I invite you to join this exciting ride, too. This book is your ticket!

Waterloo, April 2013 Krzysztof Czarnecki

Preface

The idea for this book arose from a series of lectures on modern programming paradigms, feature-oriented programming, and software product lines that are continuously held at the Universities of Magdeburg, Marburg, Passau, Texas at Austin, and others. Our collaboration reaches back to 2006, when Sven and Christian visited Don's group in Austin. Don's lecture on feature-oriented programming was inspiration for the lecture series set up in 2007 at the Universities of Magdeburg and Passau, which is the basis for this book. In a joint effort, we developed and continuously refined the teaching material for the lectures since then, until the present day.

Our interest in this topic was always the developer's perspective of discussing implementation techniques that are suitable for constructing variable software. We would have preferred to use a textbook for our lectures from the shelf, but existing product-line textbooks said precious little on implementation techniques. Eventually, in 2011, the material was stable, such that the natural next step was to write a proper text on this topic, meant not only for our students, but for all students of computer science and related fields as well as researchers and practitioners interested in software product lines.

Writing a textbook is an enormous endeavor, and this book would not have been possible without the help of our colleagues, students, friends, and families. In particular, we thank Martin Kuhlemann, Jörg Liebig, Norbert Siegmund, and Thomas Thüm, who used and improved the teaching material that was the basis for this book. Furthermore, we thank David Broneske for his support in producing proper graphics for this book, Jörg Liebig, Sandro Schulze, Norbert Siegmund, and Thomas Thüm for feedback on selected chapters and the exercises proposed in the book, as well as pure-systems, Janet Siegmund, and Thomas Thüm for providing screenshots of the tools pure::variants, FeatureCommander, and FeatureIDE.

Besides feedback and support from colleagues, we acknowledge the financial support of the German Research Foundation and the National Science Foundation for a number of research projects related to the topics covered in this book (the NSF Science of Design projects: CCF-0438786 and CCF-0724979, the ERC grant #203099, and the DFG projects: FAME-DBMS—Sa 465/32, Feature Foundation—AP 206/2, SafeSPL—AP 206/4, and Pythia—AP 206/5).

Finally, but not the least, we are grateful to our families and friends whose support was a necessary basis for the success of this endeavor.

Passau, February 2013	Sven Apel
Austin	Don Batory
Pittsburgh	Christian Kästner
Magdeburg	Gunter Saake

Contents

Part I
Software Product Lines

Chapter 1
Software Product Lines

After reading the chapter, you should be able to

- understand the key concepts and the motivation of product-line development,
- characterize product lines in the context of handcrafting and mass production,
- weigh the promises and potential drawbacks of the product-line approach, and
- discuss the specific characteristics of software product lines.

Software product lines aim at empowering software vendors to tailor software products to the requirements of individual customers. In this sense, software product lines follow a development that emerged in industrial manufacturing over the last 200 years. Starting with handcrafting of individual goods, the advent of mass production scaled the production process to large quantities, but neglected individualism, as all products were the same. With mass customization, individualism returned to the focus of attention. Manufacturers systematically planned and designed product lines to cover a whole spectrum of possible products and variations thereof, serving the individual needs and wishes of many customers. Software product lines take the same line and reconcile mass production and mass customization in software engineering.

1.1 From Individualism to Standardization and Back Again

Before the advent of mass production in the industrialization age, the manufacturing process was essentially *handcrafting*—a labor-intensive and highly individual process. Be it machines, furniture, buildings, weapons, or clothing, no two goods were produced exactly the same. Skilled craftsmen had considerable experience of what and how to build, but each product was unique in the sense that it was built from scratch. Handcrafting makes it easy to incorporate individual requirements and to build a product specifically for the customer's needs.

As a consequence of industrialization, *mass production* changed this picture fundamentally. It was one of the key driving factors of the socio-techno-economic revolution that started in the eighteenth century and whose effects resonate until the present day. Mass production based on assembly lines required standardized parts that can be produced individually and that are eventually combined to create more complex products. Mass production greatly improved productivity, compared to hand crafting: The focus on standardized products reduced production costs and improved the quality of products and processes. However, individualism in the sense that a manufacturer incorporates needs and wishes of individual customers was lost (or less important).

Today, almost a cliché, Henry Ford's expression "Any customer can have a car painted in any color that he wants as long as it is black." This statement of Ford and Crowther (1922, p. 72), has its roots in the fact that the color black dried faster. Though early Ford cars were available in many colors, for the Model T, efficiency of the newly introduced assembly line won over individualism. Similar effects could be observed in many other domains, such as mass-produced houses in suburbs instead of individual blue prints, mass-produced clothing in standardized sizes instead of tailoring-to-measure, and so forth. In all cases, prices drop, quality rises, but at the cost of having only few standardized products.

Recognizing that different customers have different needs and wishes, manufacturers started in the twentieth century to increase diversity in their product portfolios. For example, car manufacturers offer many different variations of a car at different prices, or chemical companies offer laundry detergent with different specialized fragrances and additives. Manufacturers still use reusable parts, but combine them in different ways, prepare specific alternatives for individual parts, or add extra parts. In short, the idea of a product line was born: A *product line* is a set of products in a product portfolio of a manufacturer that share substantial similarities and that are, ideally, created from a set of reusable parts. Instead of offering a single, standardized product, manufacturers aimed at diversification, this way, being able to offer multiple products tailored to individual market segments, including products for niche markets.

In many domains, *mass customization* has marked a return to individualism in production, beyond a few product variations in the manufacturer's portfolio. Nowadays, manufacturers of cars, computers, and many other products allow their customers to configure products to their needs before production. The production process still has mass-production characteristics: The products are constructed at large scale, often in an automated fashion, based on standardized, reusable parts. The goal is to share as many parts as possible between all products; however, manufacturers support variations within their production process, and they allow customers to select from alternatives, for example, different engines, colors, interior features, electronics, and so forth in the automobile industry. Customers tailor their product by choosing among a predefined set of configuration options. Often, the configuration space is incredibly huge. Today, most automobile manufacturers hardly ever produce two identically configured cars in a year. Likewise, many startup companies serve the demand for individualism and offer tailor-made clothing (shoes, trousers, bags), food (chocolate, muesli, juices), electronics, tools, medicine, and many others.

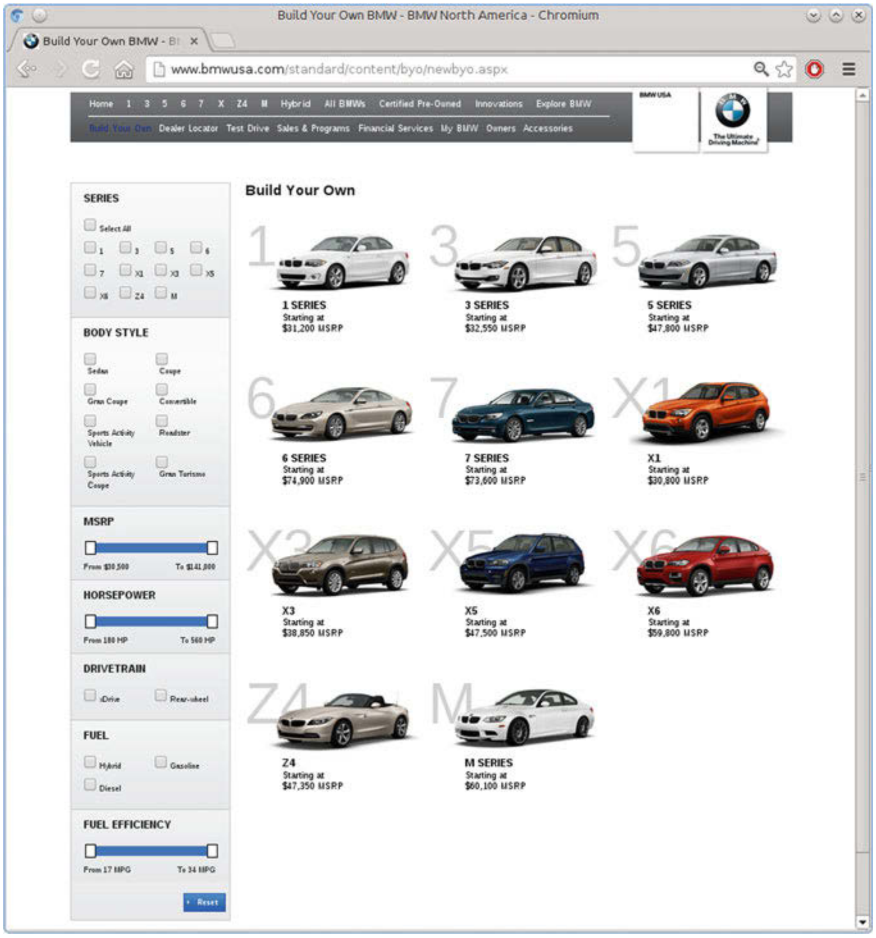

Build Your Own BMW - BMW North America - Chromium
www.bmwusa.com/standard/content/byo/newbyo.aspx
Home 1 3 5 6 7 X Z4 M Hybrid All BMWs Certified Pre-Owned Innovations Explore BMW
Build Your Own Dealer Locator Test Drive Sales & Programs Financial Services My BMW Owners Accessories
SERIES
Select All
BODY STYLE
Sedan Coupe Gran Coupe Convertible Sports Activity Vehicle Roadster Sports Activity Coupe Gran Turismo
MSRP
From $30,500 To $141,800
HORSEPOWER
From 180 HP To 560 HP
DRIVETRAIN
xDrive Rear-wheel
FUEL
Hybrid Gasoline Diesel
FUEL EFFICIENCY
From 17 MPG To 34 MPG
Reset
Build Your Own
1 SERIES Starting at $31,200 MSRP
3 SERIES Starting at $32,550 MSRP
5 SERIES Starting at $47,800 MSRP
6 SERIES Starting at $74,900 MSRP
7 SERIES Starting at $73,600 MSRP
X1 Starting at $30,800 MSRP
X3 Starting at $38,850 MSRP
X5 Starting at $47,500 MSRP
X6 Starting at $59,800 MSRP
Z4 Starting at $47,350 MSRP
M SERIES Starting at $60,100 MSRP

Fig. 1.1 BMW's car configurator

Example 1.1 Automobile product lines. As said previously, almost every car that leaves a modern automobile factory is unique—each tailored to the needs and wishes of a particular customer. But how does the customer communicate his requirements? Today, one can use Web-based car configurators for this task. In Fig. 1.1, we show a snapshot of BMW's car configurator. The customer can make various choices, including series, body style, color, power, price, efficiency, and many more.

Note that Fig. 1.1 shows only a small excerpt of the choices a customer can make. Actually, there are hundreds and thousands of possible choices. This rich set of configuration options gives rise to an astronomical number of possible car variations. It is the task of a configurator to guide the customer through a configuration, providing feedback, hide invalid choices, and so forth. It acts like a wizard that, in multiple steps, offers choices in terms of check and radio boxes; there are even sliders to choose within a spectrum of possible values.

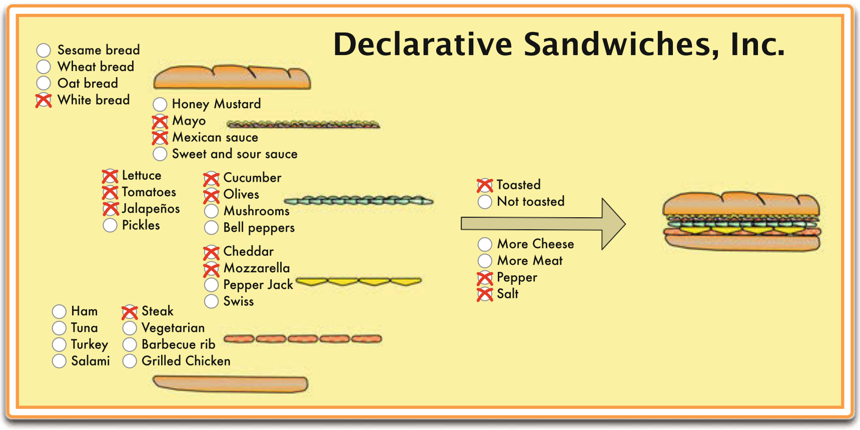


Fig. 1.2 A sandwich configurator

Of course, once a configuration is set, the corresponding car is not designed and built from scratch. Rather it is constructed automatically from reusable parts that are related to the choices made by the customer. Some choices are directly related to individual mechatronics components (for example, transmission type), others emerge from the interplay of multiple parts (for example, fuel efficiency). □

Example 1.2 Sandwich shop. An unusual yet valid and illustrative example of a successful application of a product-line approach can be found in the fastfood industry. Some sandwich shops do not sell a few predefined sandwich types, but allow their customers to choose among a set of options to create their own favorite sandwich. The space of possible choices is large, including different kinds of bread, main and side toppings, as well as various sauces and spices, but finite. The producer does not take arbitrary requests and starts thinking about shopping and preparing ingredients for each order; instead the producer prepares parts, such as sliced tomatoes, precooked meats, and sauces, that can be combined. Some companies provide even pen-and-paper configurators like the one in Fig. 1.2. Much like in the automotive example, customers cannot choose arbitrarily (for example, they can choose only one kind of bread), but are guided by a configurator—in the case of the sandwich shop, by descriptions provided on the configuration sheet. The space of possible sandwiches is huge, possibly offering a suitable and tailored choice to the vast majority of customers. □

1.2 Specialized and Standardized Software

Software development has seen a history not quite unlike that of producing physical goods. Early software products were handcrafted for specific hardware and sold, bundled with the hardware, in small numbers. Software was handcrafted by a few experts. However, software production became increasingly challenging and expensive with the demand for more and more complex software and the diversification of application scenarios.

Software producers attempted a strategy not unlike that of mass production: standardization. Instead of redeveloping operating systems, compilers, and tools again and again, the software industry agreed on standard platforms. This eventually enabled to production of shrink-wrapped software packages that were deployed on scale and could be installed on millions of Unix work stations or Windows desktops. Fitting to our analogy, these products are often called *standard software*. Developing a separate word-processing, database, or accounting software for every company would be expensive and error-prone; buying a standardized and established software solution, such as Microsoft Word, IBM DB2, or SAP R/3, *off-the-shelf* is often the more sensible choice.

However, much like mass production provides standardized goods at the cost of not supporting individualism and rare corner cases, standard off-the-shelf software aims at a *one-size-fits-all* solution. The idea is to provide software that satisfies the needs of most customers, which leads almost automatically to the situation, in which customers miss desired functionality and are overwhelmed with functionality they do not need actually (just think of any contemporary office or graphics program). It is often this generality that makes software complex, slow, and buggy.

While standardization empowered the software industry to substantially scale software development and to provide affordable software to a broad market, it often does not address smaller market segments, nor individual needs and wishes of single customers. Especially, for resource-constrained or energy-constrained environments, such as embedded systems, smart cards, and sensor networks, bloated one-size-fits-all solutions are barely suitable, which results in the situation that much software is still written from scratch, even though it may be similar to existing products. For example, Oracle bought the existing database system Berkeley DB to enter the market of data management for embedded system, instead of developing their own solution based on their existing product and their experience with developing database tools for decades. Also, specific requirements and niche market segments are typically not well-addressed by off-the-shelf software; this includes novel algorithms for specific problems such as image recognition, high-performance computing, company- and market-specific business models, and so on. Though not required by the masses, specific solutions can provide competitive advantages for individual companies, customers, and institutions.

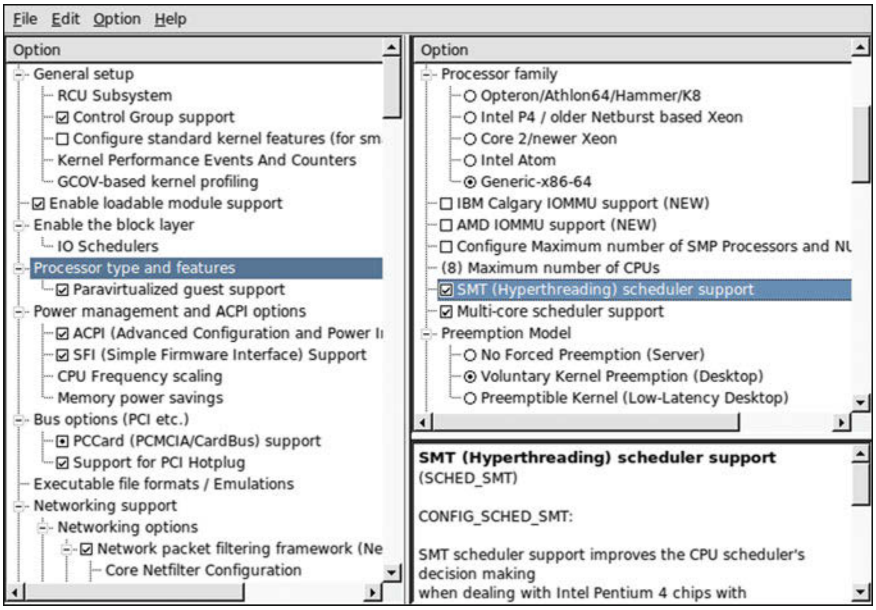
1.3 Software Product Lines

Already since the late 1960s, and even more in the 1990s, *software product lines* have gained momentum in the software industry. Instead of developing software systems from scratch, they should be constructed from reusable parts. Instead of composing a software system always in the same way, it should be tailored to requirements of the customer, where customers can select from a large space of configuration options.

The software product line approach provides a form of mass customization by constructing individual solutions based on a portfolio of reusable software components. It introduces individualism into software production, but still retains the benefits of mass production in that whole domain and market segments can be served. The need for individualism arises from different requirements on the software regarding functionality, target platforms, and nonfunctional properties, such as performance and energy consumption.

Example 1.3 Operating-systems. Almost every computing system today contains an operating system, which mediate between hardware and application software. Modern operating systems have to run on different hardware platforms and serve the needs of various applications. For example, the Linux kernel runs on a wide variety of different platforms, including embedded devices, desktop systems, and large-scale servers, supporting a huge portfolio of different applications scenarios, from sense-and-control applications, over office software and games, to high-performance computing and server software.

Clearly, there cannot be a single operating system that efficiently supports all kinds of different platforms and application scenarios. Instead, modern operating systems are configurable. In fact, systems like Linux kernel are software product lines (Sincero et al. 2007). Much like in the automotive domain (Fig. 1.1), users can select among a large set of options (up to 10 000 features (Tartler et al. 2011)) to tailor the Linux kernel to their needs. In Fig. 1.3, we show a screenshot of Linux's configuration tools, called Kconfig, that can be used for this task. ☐



```
Option
- General setup
    RCU Subsystem
    ☑ Control Group support
    ☐ Configure standard kernel features (for sm
    Kernel Performance Events And Counters
    GCOV-based kernel profiling
  ☑ Enable loadable module support
- Enable the block layer
    IO Schedulers
- Processor type and features
    ☑ Paravirtualized guest support
- Power management and ACPI options
    ☑ ACPI (Advanced Configuration and Power I
    ☑ SFI (Simple Firmware Interface) Support
    CPU Frequency scaling
    Memory power savings
- Bus options (PCI etc.)
    ⊡ PCCard (PCMCIA/CardBus) support
    ☑ Support for PCI Hotplug
  Executable file formats / Emulations
- Networking support
  - Networking options
    - ☑ Network packet filtering framework (Ne
        Core Netfilter Configuration
```

```
Option
- Processor family
    ○ Opteron/Athlon64/Hammer/K8
    ○ Intel P4 / older Netburst based Xeon
    ○ Core 2/newer Xeon
    ○ Intel Atom
    ◉ Generic-x86-64
  ☐ IBM Calgary IOMMU support (NEW)
  ☐ AMD IOMMU support (NEW)
  ☐ Configure Maximum number of SMP Processors and NU
  (8) Maximum number of CPUs
  ☑ SMT (Hyperthreading) scheduler support
  ☑ Multi-core scheduler support
- Preemption Model
    ○ No Forced Preemption (Server)
    ◉ Voluntary Kernel Preemption (Desktop)
    ○ Preemptible Kernel (Low-Latency Desktop)
```

```
SMT (Hyperthreading) scheduler support
(SCHED_SMT)

CONFIG_SCHED_SMT:

SMT scheduler support improves the CPU scheduler's
decision making
when dealing with Intel Pentium 4 chips with
```

Fig. 1.3 Linux's kernel configuration tool

1.4 Promises of Software Product Lines

In the tension between developing individual software products from scratch and developing standardized one-size-fits-all products, software product lines promise distinct benefits , of which we discuss next the most important ones:

Tailor-made. A product-line approach to software production facilitates tailoring products to individual customers. Instead of providing a standardized product (or a small set of preconfigured products, such as Community, Professional, and Enterprise editions), a software vendor can produce a whole set of differently tailored products.
Reduced costs. While providing each customer its desired solution, product-line vendors do not need to pay the cost of designing and developing each product from scratch. Instead, they develop reusable parts that can be combined in different ways. The development cost per product per customer can be reduced to selecting which parts to combine (potentially add missing ones) and to testing the resulting product. While the upfront investment required for such an approach is certainly larger than developing a single software product (we need to design reusable parts and implement variations that might not be required for the first product), the approach pays off in the long term, when multiple tailored products are requested. We illustrate the economic promise of software product lines in Fig. 1.4.[1]
Improved quality. Industrial mass production has improved quality because standardized parts can be checked systematically and tested in many products. Standardized software offer similar benefits, compared to software developed individually from scratch. Software product lines offer a compromise, in that different products

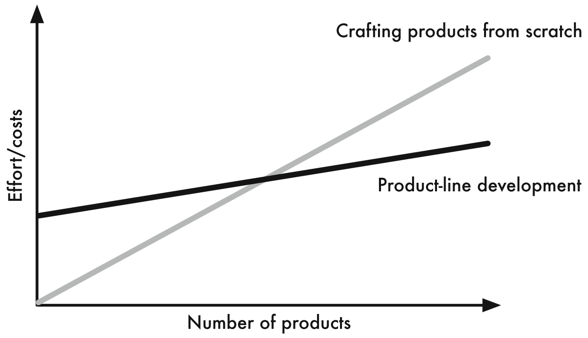


Fig. 1.4 Effort/costs of crafting products individually versus product-line development

[1] When comparing production costs of software to industrial goods, be aware of a common pitfall: The costs for replicating software are near zero (and is still low when considering distribution). A cost reduction in mass production of standard software does not result from a more efficient assembly process (which a build script can do for free), but lies mainly in the one-time design and implementation effort that can serve many customers, which all receive the same product. When talking about costs in software product lines, we talk about development costs, which are similar to design and planning costs in industrial manufacturing.

are constructed from standardized parts. Though not every combination is used by millions of customers, parts can be standardized and checked in isolation to some degree, and parts are reused and tested in multiple products. Especially, parts that are used frequently lead to more stable, lean, and reliable products, than handcrafting.
Time to market. While standard software is readily available, handcrafted software products require significant costs and, sometimes more importantly, time, before they can be released. If a customer selects only from predefined configuration options (that map to existing parts), a software vendor can quickly produce the corresponding software product by assembling the existing, corresponding parts. Even if a customer requests functionality that has not been prepared, building a new product on top of existing well-designed, reusable parts is much faster than developing it entirely from scratch. A well-designed platform that can be extended for new products promises the possibility to quickly *react to market changes*, more so than standardized software or individual development could.

As said previously, product-line development (for software and physical goods) comes at a price. Preparing reusable parts requires a significant upfront investment (see Fig. 1.4). Furthermore, developers start designing multiple potential products at the same time, which raises complexity. Often, offering various configuration options gives rise to an exponentially large configuration space that no single person can oversee entirely. Variability management (deciding which parts to prepare) becomes necessary, and requires additional effort.

1.5 Success Stories

To illustrate the practical relevance of software product lines, we summarize a few of the success stories:[2]

Boeing. Boeing develops a product line of operational flight programs, which are mission-critical, distributed real-time, and embedded applications supporting the avionics and cockpit functions for the pilot. Carefully designed approaches to handle commonality and variability are the crucial success factors of this product line.
Bosch. Bosch develops a product line of engine-control software for gasoline systems. In this domain, developers face extreme variability, permanently growing complexity, and high pressure on cost. Bosch considers the product-line approach as the key to produce suitable software products and to enter new markets.
Hewlett Packard. Hewlett Packard pursues a product-line approach to develop printer firmware. It consists of over 2000 features supporting hundreds of printers. Moving to a product-line approach, products could be produced using 1/4 of the staff, in 1/3 of the time, and with 1/25 the number of bugs of earlier products.
Toshiba. Toshiba produces power generation, transmission, and distribution equipment, supporting customers with capabilities ranging from power plant construction

[2] The first three have been assembled and honored in the Product-Line Hall of Fame: `http://splc.net/fame.html`.

to management and operation. In this context, Toshiba developed a software product line that separates variable and non-variable software components and provides a fill-in-the-format-type configurator to support individual management of variable parts and software logic parts across the period from manufacturing to system maintenance.

General Motors. General Motors develops the control software for their powertrains as a software product line. It supports various electronic architectures (5 engine-control modules, 4 transmission-control modules, and 3 powertrain-control modules) and physical architectures (diesel engines verses gas engines, clutch-to-clutch transmissions verses freewheel transmissions). Today, the software product line is the basis for nearly all new control modules being developed by General Motors.

1.6 A Feature-Oriented Approach

Software product lines facilitate the industrialization of software development. Ideally, based on a set of reusable parts, a software manufacturer can generate a software product based on the requirements of a certain customer. The concept of a feature is central to achieve this level of automation. Features are used to distinguish the products of a product line, for example:

- "My text editor provides a spell-checking feature."
- "Database system A provides multi-user support, Database B does not."
- "E-mail client A supports IMAP and POP3, client B supports only POP3."
- "The game we are going to develop shall run on Android and Windows."
- "Both financial software products support international transactions."

In some sense, features bridge the gap between the requirements a customer has and the functionality a product provides. The key idea of *feature orientation* is to organize and structure the whole product-line process as well as all software artifacts involved in terms of features. This way, it is easy to trace the requirements of a customer to the software artifacts that provide the corresponding functionality. In technical terms, a feature-oriented approach makes features explicit in requirements, design, code, testing, and so forth—across the entire life cycle. It is a key goal of this book to introduce and discuss techniques to implement feature-oriented product lines based on this premise.

1.7 Running Examples

To tie this book together, we use two running examples that are well-documented in the product-line literature: data management for embedded systems and variations of graph data structures.

Table 1.1 Data management in automotive systems

Subsystem	Persistence	Recovery	Consistency	Queries	Granularity
Navigation system	✓	✓		SQL	Database
Driver's logbook	✓	✓	✓	Cursor	Tables
Total distance recorder	✓	✓	✓	Fetch	Tuple
Number of revolutions recorder	✓			Integer	

Example 1.4 Data management for embedded systems. State-of-the-art *relational database systems* support a rich set of features such as multi-user operation, high-level query languages, and powerful query optimization. However, they are much too complex and heavyweight to be used in embedded systems, such as in sensors networks or mobile devices. Still, features such as persistence, recovery, and index structures are needed in embedded systems, too. The inability to size-down fully-fledged database systems by stripping and replacing unnecessary features led to separate development lines of data management for embedded systems, henceforth called *embedded data management*.

Storing data is at the heart of every product in the domain of embedded data management, although different products may support different data types and storage structures. Support for transactions and recovery are typical, but may not be required in all application scenarios. Providing a single implementation for all scenarios is infeasible, because of the overhead of unused code on systems with restricted resources. In Table 1.1, we show different requirements for data management used in embedded automotive systems.

Embedded data management is a perfect candidate for using a product-line approach. Leich (2012) provides a comprehensive overview of the state of the art in this field. In this book, we refer to two research prototypes of product lines for embedded data management: FameDBMS (Rosenmüller et al. 2008) and the feature-oriented refactoring of Berkeley DB (Rosenmüller et al. 2009a), which we introduce in the respective chapters. □

Example 1.5 A product line of a graph library. To illustrate variability implementation techniques at a technical level, the scenario for embedded data management is still too complex, and domain concepts would distract from technical issues (How to implement a B-tree?). Hence, we use a simple product line of graph data structures for this task. It was introduced by Lopez-Herrejon and Batory (2001) as a standard problem to discuss and compare product-line techniques and has been used in hundreds of publications since. The core of its implementation fits on less than a page. In practice, though, this example would mostly like not be target of a product-line approach because of its simplicity.

At the source-code level, we use the graph library as our main technical example throughout the book. It constitutes a product line of implementations of graph data structures and algorithms. The base implementation language is Java. The listing in Fig. 1.5 gives an impression of how graph data structures with nodes and edges

```java
class Graph {
  Vector nodes = new Vector():
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n,m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  void print() {
    for(int i=0; i<edges.size(); i++){
      ((Edge) edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}
```

```java
class Node {
  int id = 0;
  Node (int _id) { id = _id; }
  void print() {System.out.print(id);}
}


class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {a=_a; b=_b;}
  void print() {
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
  }
}
```

Fig. 1.5 Graph library: an implementation example

are realized. Although small, there is a considerable number of features to vary: vertices and edges can be colored, edges directed or undirected, edges can be stored as separate objects or adjacency lists, and so forth. One can also choose among a diverse selection of algorithms that work on graphs in different configurations, such as detecting cycles, searching shortest paths, and computing the minimal spanning trees.

In addition to the features that have an effect on externally visible functionality, there is also variability in the internal implementation, for example, how edges are represented (such as, using explicit objects or implicit links between vertices). For instance, as an alternative to the implementation in Fig. 1.5, we could store adjacent nodes inside class `Node`. □

1.8 Intended Audience of the Book

In the last 15 years, several books have discussed issues of software product lines. How is this book different?

We provide a special perspective on software product lines: We take a developer's viewpoint that focuses on the development, maintenance, and implementation of product-line variability. We blend out most management issues, such as requirements analysis, scoping and portfolio management, and team organization. The concept of a feature pervades the entire life cycle, including design, implementation, and validation and verification. In short, *features are a central concept in all phases of product-line development. Furthermore, we concentrate on automated product derivation based on a user's feature selection.*

As a result, this book is unique for the following reasons:

- Features are central to variability. We introduce feature models in Chap. 2 and use features throughout remaining chapters to guide product-line development.

- Software product lines with a huge number of products require systematic reuse of code artifacts across products. We focus on *automatic product derivation*, where a specific product, specified by a feature selection, can be generated automatically from reusable artifacts. These artifacts include both code and other supplementary documents.
- We are not biased toward a single implementation technique or programming paradigm. In fact, we broadly survey different implementation mechanisms and discuss their trade-offs. Besides, novel programming paradigms such as feature-oriented programming (which still has to find its way into industrial practice), we discuss in equal depth classic approaches, including preprocessors and frameworks. In fact, the presentation of implementation techniques is the core of this book (Part II).
- We cover a set of advanced topics that include novel developments, cutting-edge research, and open issues in the areas of feature interactions as well as product-line refactoring and analysis.
- Methods and tools for supporting engineers are crucial in modern software development. Almost all chapters describe the current state of tool support for the discussed tasks (programming, code generation, code analysis, and so forth). In Appendix A, we provide an overview of current systems supporting the engineering of software product lines.

1.9 How to Read this Book

This book consists of three parts. Part I provides a gentle introduction of feature-oriented software product lines.

- Chapter 1 motivates the product-line approach and provides an overview of the book.
- Chapter 2 introduces the product-line development process, consisting of domain and application engineering.

Part II is the core of this book. It covers a wide variety of implementation techniques for software product lines. A reader interested in a particular implementation technique may directly jump to the corresponding chapter, after reading Chap. 3.

- Chapter 3 introduces basic concepts, dimensions of classification, and quality criteria for product-line implementation techniques.
- Chapter 4 reviews classic implementation techniques used in product-line development, including run-time and compile-time parameters, design patterns, frameworks, and components.
- Chapter 5 describes a wide array of classic tools used in product-line development, for example, build systems, preprocessors, and version control systems.
- Chapter 6 discusses advances in programming languages to support product-line development, especially, with regard to feature modularity.

- Chapter 7 explains advanced tool support for the development of software product lines, including the concept of virtual separation of concerns.

Part III is devoted to advanced topics related to feature-oriented product lines.

- Chapter 8 discusses the challenges and promises of refactoring of feature-oriented product lines.
- Chapter 9 is concerned with the interaction between features, which need to be coordinated properly.
- Chapter 10 describes techniques and tools for the analysis of feature-oriented product lines.

Finally, in Appendix A, we provide a list of product-line tools, along with a description of how they relate to the topics covered in this book.

1.10 Further Reading

The field of software product lines started its success story in the 1990s, though the field has its roots in the much earlier works on program families (McIlroy 1969; Parnas 1979). So far, a number of foundational books about software product lines have been published, most notably the books by Clements and Northrop (2001) and Pohl et al. (2005). They cover all facets of product-line engineering, but provide only limited material on variability implementation techniques. Besides these foundational books, several authors summarize their experience of applying software product lines to practice, for example, van der Linden et al. (2007) and Kang et al. (2009).

Another collection of books is concerned with techniques to implement software product lines, including generative programming (Czarnecki and Eisenecker 2000), software architecture (Bosch 2000), aspect-oriented, and model-driven engineering (Rashid et al. 2011). In contrast, we do not focus on a single implementation technique or programming paradigm, but we introduce and discuss a whole spectrum of implementation techniques, with mutual strengths and weaknesses.

Chapter 2
A Development Process for Feature-Oriented Product Lines

After reading the chapter, you should be able to

- define the relevant terms: product line, feature, feature selection, feature dependency, product, domain,
- understand why a product line targets a specific domain,
- explain the product-line development process consisting of domain engineering and application engineering (including how the different phases interact),
- distinguish problem space and solution space,
- understand what drives scoping decisions,
- explain the economic lever of product lines and understand the benefit of automation,
- model features and feature dependencies by means of feature models,
- translate feature diagrams to propositional formulas, and
- discuss trade-offs between different adoption paths.

In this chapter, we introduce basic concepts that arise in the engineering of feature-oriented software product lines. We narrow down the term *feature*, introduce an overall development process, and illustrate how to model and formalize variability in product lines.

2.1 Features and Products

Features are the concerns of primary interest in product-line engineering. The concept of a feature is inherently hard to define precisely as it captures, on the one hand, intentions of the stakeholders of a product line, including end users and, on the other, design and implementation-level concepts used to structure, vary, and reuse software artifacts. Consequently, there are many definitions, below ordered from abstract to technical (Classen et al. 2008):

1. Kang et al. (1990): "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"
2. Kang et al. (1998): "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"
3. Czarnecki and Eisenecker (2000): "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"
4. Bosch (2000): "a logical unit of behavior specified by a set of functional and non-functional requirements"
5. Chen et al. (2005): "a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"
6. Batory et al. (2004): "a product characteristic that is used in distinguishing programs within a family of related programs"
7. Classen et al. (2008): "a triplet, $\mathtt{f} = (\mathtt{R}, \mathtt{W}, \mathtt{S})$, where $\mathtt{R}$ represents the requirements the feature satisfies, $\mathtt{W}$ the assumptions the feature takes about its environment and $\mathtt{S}$ its specification"
8. Zave (2003): "an optional or incremental unit of functionality"
9. Batory (2005): "an increment of program functionality"
10. Apel et al. (2010): "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"

The first seven definitions treat features mainly as a means to communicate between the different stakeholders of a product line (end users, managers, programmers, and so forth), in order to distinguish software products. The last three definitions treat features as design decisions and implementation-level concepts that are part of the software construction phase. These different views on features stem, of course, from the different use of features in the different phases of product-line engineering. To capture the essence and commonalities of prior usage, we define features as follows:

Definition 2.1 A *feature* is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle. □

The product portfolio of a product line is defined by its features and their relations. A specific product is identified by a subset of features, called a *feature selection*. Not all feature selections are valid and specify meaningful products. As we saw in the previous chapter, the features Toasted and Not toasted of a sandwich are mutually exclusive; so too are the exterior car trims Standard, Luxury, Premium, and Platinum. A constraint on the feature selection is called a *feature dependency*. Feature dependencies are modeled explicitly in product lines as part of feature modeling, which we discuss later.

Definition 2.2 A *product* of a product line is specified by a valid feature selection (a subset of the features of the product line). A feature selection is *valid* if and only if it fulfills all *feature dependencies*. □

Features, feature selections, feature constraints, and products arise in all kinds of product lines, and are not limited to software product lines. In the following sections, we discuss the role of features in the product-line engineering process. We introduce feature models as a formalism to describe features and their constraints. Finally, a translation of feature model to propositional logic opens the door for formal methods of analyzing product-line variability.

2.2 A Process for Product-Line Development

Most processes of traditional software engineering target the life cycle of a single software system. Independent of the specifics of the process used, developers collect requirements for the target system, and design and implement the system, either in separate, consecutive phases or in agile cycles. For software product lines, we must change our way of thinking about software development. In contrast to analyzing and implementing a single system, we have to look at a variety of desired systems that are similar but not identical.

A key success factor of product-line development is to set a proper focus on a particular, well-defined and well-scoped domain.

Definition 2.3 A *domain* is an area of knowledge that:

- is scoped to maximize the satisfaction of the requirements of its stakeholders,
- includes a set of concepts and terminology understood by practitioners in that area,
- and includes the knowledge of how to build software systems (or parts of software systems) in that area.

(Adopted from Czarnecki and Eisenecker (2000), p. 34) □

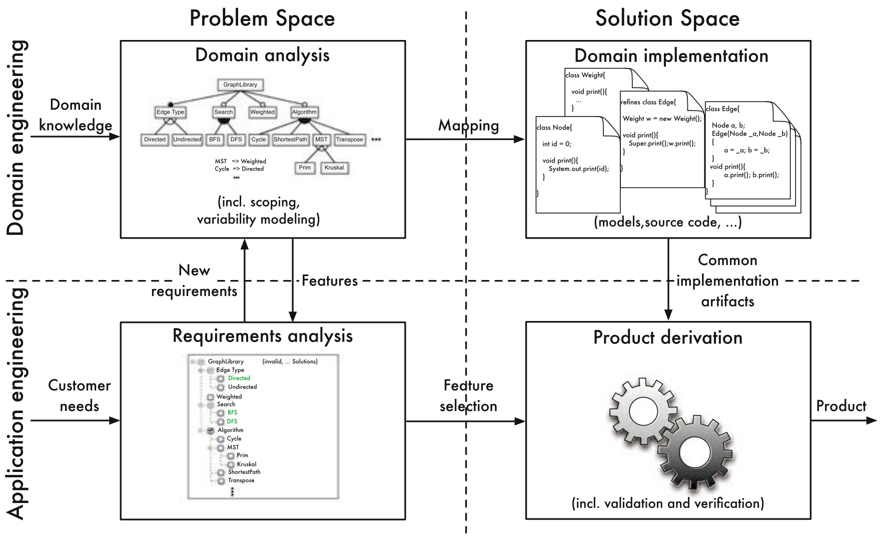


Fig. 2.1 Overview of an engineering process for software product lines

In the past, software product lines have been developed for a wide variety of domains, including operating systems, database systems, middleware, automotive software, compilers, healthcare applications, and many more.

The broader the domain of a product line is the larger is the number of possible stakeholders' requirements that can be covered in the form of individually tailored products. However, the broader the domain, the smaller is the set of similarities among products. For example, the domain of system software is huge, which includes operating systems, drivers, network software, database systems, and many more. Although there are similarities that could be exploited in system software, individual systems have substantial differences, which decrease potential for reuse. Focusing on the (sub)domain of database systems or even embedded database systems, increases the reuse potential, while keeping maintenance effort acceptable. The bottom-line is that a proper scoping of the target domain is essential, as we discuss further in Sect. 2.2.1.

A development process for software product lines has to take these peculiarities into account. Two issues play a crucial role: the explicit handling of *variability* and the systematic *reuse* of implementation artifacts. For both, an appropriate *structuring* of process and software artifacts is imperative.

The specific characteristics of software product lines lead to a separation between domain engineering and application engineering and between problem space and solution space. In Fig. 2.1, we illustrate a two-dimensional structure with four clusters of tasks in product-line development and the mappings between them, which we explain next.

Domain engineering (top half of Fig. 2.1) is the process of analyzing the domain of a product line and developing reusable artifacts. Domain engineering does not result in a specific software product, but prepares artifacts to be used in multiple, if not all, products of a product line. Domain engineering targets *development for reuse*. In contrast, *application engineering* (bottom half of Fig. 2.1) has the goal of developing a specific product for the needs of a particular customer (or other stakeholder). It corresponds to the process of single application development in traditional software engineering, but reuses artifacts from domain engineering where possible. It targets *development with reuse*. Application engineering is repeated for every product of the product line that is to be derived.

The distinction between the *problem space* and *solution space* highlights two different perspectives. The problem space (left half of Fig. 2.1) takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space. In contrast, the solution space (right half of Fig. 2.1) represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic reuse.

The orthogonal distinctions between domain and application engineering as well as problem and solution space give rise to four clusters of tasks in product-line development:

- *Domain analysis* is a form of requirements engineering for an entire product line. Here, we need to decide the *scope* of the domain, that is, decide which products should be covered by the product line and, consequently, which features are relevant and should be implemented as reusable artifacts. The results of domain analysis are usually documented in a feature model.
- *Requirements analysis* investigates the needs of a specific customer as part of application engineering. In the simplest case, a customer's requirements are mapped to a feature selection, based on the features identified during domain analysis. If novel requirements are discovered, they can be fed back into domain analysis, which may result in a modification of the feature model (and the reusable domain artifacts).
- *Domain implementation* is the process of developing reusable artifacts that correspond to the features identified in domain analysis. Although there are many kinds of artifacts relevant in software product lines (including design, test, and documentation artifacts), we concentrate in the book on implementation artifacts, in particular, source code. Nevertheless, the basic ideas and techniques apply also to non-code artifacts. Depending on how variability is implemented (see Part II of this book), developers might produce very different artifacts in this step, from run-time parameters and preprocessor directives to plug-ins and components, and many more.
- *Product derivation* (or *product generation* or *product configuration* or *product assembly*) is the production step of application engineering, where reusable

artifacts are combined according to the results of requirement analysis. Depending on the implementation approach, this process can be more or less automated, possibly, involving several development and customization tasks.

As said previously, domain engineering is performed once for the entire product line, whereas application engineering is performed for every individual product. A goal of product-line development, in general, is to move development effort as much as possible from application engineering to domain engineering. For example, if quality assurance (such as code inspections) can be done in domain engineering instead of investigating individual products, costs can be dramatically reduced. This way, shared artifacts are investigated only once. The more we evolve application engineering into a series of generation tasks, the smaller the costs per product become (cf. Fig. 1.4). A major goal of feature-oriented product lines is to fully automate product derivation.

In the following sections, we look at the four quadrants of development tasks of Fig. 2.1 in detail and provide examples from our application scenarios. We focus primarily on the problem-space tasks, whereas we go into detail about the solution space in Part II of this book.

2.2.1 Domain Analysis

Domain analysis is a form of requirements analysis for the whole product line. It is concerned with the problem space. It contains two primary tasks: domain scoping and domain modeling.

Domain Scoping

Domain scoping is the process of deciding on a product line's extent or range. Typically, management decides which of all possible requirements arising in a domain should be considered. The scope describes desired features or specific products that should be supported. For example, the focus may be on embedded database systems for a particular hardware architecture, instead of supporting multiple architectures. During domain scoping, domain experts collect information about the target domain, for example, by analyzing handbooks, existing systems, interviews with domain experts, potential customers, and so on.

As said previously, product lines with a small scope are easier to develop and maintain, as they target a well-defined domain of very similar products with few variations and much reuse. The broader the scope and the more features the product line has, the more possible customers can be satisfied. So, there is a trade-off between implementation effort and potential use of the product line. The trade-off requires careful business consideration, including determining prospective revenue, potential customers, and costs of additional features.

Scoping decisions are design decisions depending on the goals of the company developing a software product line. As such, they are typically subjective and based on previous experience.

Example 2.1 In the domain of embedded data management, a product line should cover basic data management functionalities targeting different operating systems for embedded devices. Management identifies transactions, recovery, encryption, basic queries, and data aggregation as the features that are most frequently requested. With these features, management estimates that half of all scenarios in embedded data management can be covered.

By focusing on embedded systems, several features are obviously outside the scope of the product line: Neither SQL-style query optimization nor remote storage outside the device (for example, cloud storage) are considered.

However, not all cases are so clear cut. For example, management considers whether to include security (user and access management) as an optional feature. It might open the product line for application scenarios in which customers need multi-user support, but in the considered market segment this might not be many customers. After conferring with potential customers, management decides to exclude the feature from the product line's scope, as the potential revenue does not cover the required implementation and maintenance costs. □

Domain Modeling

The scope of a product line should be recorded. *Domain modeling* captures and documents the commonalities and variabilities of the scoped domain. As a first approximation, stakeholders could give examples for possible products, as well as counter examples documenting which products are and which are *not* in the scope of the product line. Typically, commonalities and differences between desired products are identified and documented in terms of features and their mutual dependencies—the topic of *feature models* in Sect. 2.3.

Example 2.2 For the domain of embedded data management, we identify Storage, Transactions, OperatingSystem, Encryption as features to support. Only Storage and OperatingSystem are mandatory, all other features are optional. An example for restricting the possible products is that we cannot select more than one supported operating system at the same time. □

Example 2.3 For the graph library, we consider feature GraphLibrary as the base feature. Then, we have two features DirectedEdges and UndirectedEdges, which are mutually exclusive. Furthermore, we could think of features Search, either breadth-first search (BFS) or depth-first search (DFS), Weighted for weighted edges, and Algorithms. As algorithms, we consider multiple optional features that are used in many but not all applications, such as Cycle detection, ShortestPath, minimal spanning trees (MST), and Transpose. □

2.2.2 Requirements Analysis

Requirements analysis in product-line engineering is similar to requirements analysis in traditional software engineering. Requirements analysts solicit the customer's requirements, typically, using well-known requirement-analysis techniques, such as interviews and document analysis (Clements and Northrop 2001; Pohl et al. 2005). But, in product-line engineering, we can build on the knowledge gathered during domain analysis. There, we already identified possible requirements arising in the domain, so requirements analysts try to map the customer's requirements to those identified earlier during domain analysis. Ideally, requirements analysis can be reduced to the selection of existing features, such that a product can be assembled using reusable implementations artifacts associated with these features. If a customer's requirement cannot be mapped to one or more existing features, several strategies are possible:

- We can decide that the requirement is out of scope of the product line, so we simply cannot provide a corresponding feature or product.
- We can assemble the next best product without this feature and manually extend the resulting product with custom extensions. This way, we invest additional implementation effort during application engineering, which is not integrated back into the product line.
- Finally, we can decide to change the scope of our product line and include the additional requirement in the form of a new feature or changes to existing features, including domain artifacts. That is, we go back to domain engineering and implement a new feature or modify exiting ones. Subsequently, we can map the customer's requirement to these features, of which also other customers can benefit.

Again, which path to take is a business decision that must be weighed. Additional development in application engineering to patch up a product is certainly cheaper in the short run than developing a new feature available for all products (which involves again domain scoping and modeling steps), but other products of the product line cannot benefit from that development.

Example 2.4 Suppose we have a database product line with features for SQL-query processing and transaction support. If, during requirements analysis, we learn that a customer wants to use the requested product in an SQL-based, multi-user, client-server environment, we can map these requirements directly to the existing features for SQL-query processing and transaction support and their respective implementations. □

Example 2.5 Suppose a user needs a variation of the graph library that supports Dijkstra's algorithm to compute shortest paths. We can map this requirement only partially to the selection of the features for undirected and weighted edges. Dijsktra's algorithm needs to be implemented from scratch, and it is up to the product-line developer or vendor to decide whether it will be provided only to the requesting user, or if it will by also propagated back to domain engineering, thus becoming available to other users, as well. □

The identification of desired features cannot always be accomplished in a single step. In complex scenarios, when multiple stakeholders are involved, features have to be selected in multiple, consecutive steps—a process called *staged configuration* (Czarnecki et al. 2005b).

Example 2.6 Suppose we want to construct a specific product for sensor data management. In a first step, we may decide about the basic technical requirements such as choosing the operating system. Depending on the kind of data collected by the sensor, we choose, in a second step, the corresponding data types and query primitives (aggregation of values). Finally, in a third step, some non-functional requirements concerning performance, response time, footprint, and security will guide us to finish the configuration, for example, which index data structure to use. □

2.2.3 Domain Implementation

After identifying features, we want to implement them in the form of reusable artifacts. Domain (feature) implementation targets the solution space, as we are now using the developer's vocabulary to implement solutions to a customer's requirements. The implementation process comprises several considerations beyond just writing code.

First, we need to select a general implementation strategy, also known as a *reuse framework*. For example, we could use a preprocessor to include or exclude variable code conditionally or build a framework with a number of plug-in that can be combined on demand. We discuss different implementation strategies and their trade-offs in Part II of this book, therefore, we do not go into detail here.

Second, depending on the implementation strategy, we might need to prepare the design and code such that we can hook feature implementations. For example, we design how to structure common parts of the implementation and where to leave extension points and how to enable or disable extensions for features.

Besides code fragments to be reused, we also consider other kinds of documents including various models, documentation, and test cases. All these other artifacts are subject of the domain implementation, too. This is rooted in a *principle of uniformity*, as discussed in Chap. 3.

Example 2.7 For a product line for embedded data management, we choose feature-oriented programming as implementation technique (see Sect. 6.1, p. 130). We implement a basic database engine using Java and encode each feature using a separate feature module (which define additions to and modifications of the basic engine). Much like plug-ins, feature modules can be included or excluded depending on a users feature selection. □

2.2.4 Product Derivation

Depending on the choice of the implementation strategy, a product for a given feature selection (from requirements analysis) can be generated or composed using the artifacts developed in domain implementation. Most implementation strategies discussed in this book support a *fully automated* generation based on a feature selection and reusable artifacts—that is, a *push-button approach*. For example, we select the artifacts that correspond to selected features and call a composition engine to combine them into an executable, without further manual intervention.

Alternatively, we could assemble each product manually from reusable artifacts. That is, many parts of the implementation have been prepared during domain implementation and can be reused, but the combination of the artifacts is a (typically tedious) manual task, in which developers still have to write glue code to connect the artifacts and to patch up the gaps for which no reusable artifacts exist.

In both cases (automatic and manual), the resulting product usually has to be validated (or verified) before being delivered to a customer, potentially by running automated unit tests derived from artifacts provided during domain engineering. Product validation and verification is usually the last step in application engineering—more on this issue in Chap. 10.

Ideally, feature selection is the only manual activity in application engineering. The quest for automating product derivation is motivated by several promising benefits. Automating product derivation almost entirely eliminates the costs of product derivation (cf. Fig. 1.4). Also, instead of manually crafting a set of preconfigured products, automation allows products to be tailored to many individual use cases. Finally, when evolving a shared artifact (for example, to fix a bug), we can simply regenerate all products. We have a single code base instead of scattered handwritten code per product.

Example 2.8 Since our database product line is implemented using feature-oriented programming, we can automate the assembly process. For a given feature selection, we locate corresponding artifacts bundled in feature modules and compose them with fully automated tools. How this can be done is the subject of Part II of this book. □

2.3 Feature Modeling

Modeling variability is a crucial step in product-line development. There are many different approaches of variability modeling, each with a slightly different focus and goal. A common approach is to express variability in terms of common and optional features, a process called appropriately enough *feature modeling*. We use *feature models* and their graphical representation as *feature diagrams*, because they are currently the most popular form of variability models. Other examples of variability models are decision models (Schmid et al. 2011) or orthogonal variability models (Pohl et al. 2005), Czarnecki et al. (2012) discuss the differences.

Feature modeling takes place in domain analysis, but its results play a central role in other phases of product-line development, for example, for requirements analysis and product derivation—which is the reason why we provide substantial room for explanations and discussions. *In short, a feature model documents a product line's variability*. It specifies the set of valid products. Besides introducing the graphical language of feature diagrams, we also connect the graphical representation to a formal representation that is the basis of engineering tools.

2.3.1 Feature Models

A *feature model* documents the features of a product line and their relationships.

Let us start with the features. As described in Sect. 2.1, a feature represents typically a domain abstraction. We refer to a feature by its name. However, a feature is more than a name; in domain modeling, one has to look at other properties of features. Here is an (incomplete) list of potential information that domain experts can collect on features:

- Description of a feature and its corresponding (set of) requirements
- Relationship to other features, especially hierarchy, order, and grouping
- External dependencies, such as required hardware resources
- Interested stakeholders
- Estimated or measured cost of realizing a feature
- Potentially interested customers and estimated revenue
- Configuration knowledge, such as 'activated by default'
- Configuration questions asked during the requirements analysis step
- Constraints, such as "requires feature X and excludes feature Y"
- All kinds of behavioral specifications, including invariants and pre- and postconditions
- Known effects on non-functional properties, such as "improves performance and increases energy consumption"
- Rationale for including a feature in the scope of the product line
- Additional attributes, such as numbers and textual parameters, used for further customization during product generation
- Potential feature interactions

In this book, we concentrate on configuration knowledge and feature implementation.

Features are not always freely combinable. Not all features may be compatible, and some features may require the presence of other features (for example, "A must always be selected" and "B implies C"). Therefore, a feature model describes relationships between features and defines which feature selections are *valid*.

In its simplest form, a feature model comprises a list of features and an enumeration of all valid feature combinations. However, such enumeration quickly becomes too large to be practical; therefore, other notations to describe relationships have been proposed.

In principle, very different modeling approaches can describe the relationships between features. One may follow a linguistic perspective using ontologies, or a direct logic-based approach specifying the valid feature combinations using propositions. Other approaches may use known modeling formalisms such as UML and provide only a special interpretation.

In feature-oriented design and implementation, *feature diagrams* are a standard visual representation, whose semantics is specified by a translation into *propositional logic*. Feature diagrams define a feature model as a hierarchy of features and constraints among them.

2.3.2 Feature Diagrams

A *feature diagram* is a graphical notation to specify a feature model. It is a tree whose nodes are labeled with feature names. Different notations convey various parent–child relationships between features and their constraints.

If a feature `f` is a child of another feature `p`, `f` can be selected only when `p` is also selected. Typically, a feature diagram includes mutual relations between features. For example, the parent feature denotes a more general concept and the child a specialization.

Mandatory and optional features are distinguished by a small circle on the child node—a filled bullet denotes a mandatory feature, whereas an empty bullet denotes an optional feature (see Fig. 2.2). The parent node is labeled with `p`, the child node with `f`.

Specific graphical elements define additional constraints, if the child features of a common parent cannot be selected independently. Figures 2.3 and 2.4 show graphical notations for disjunctive combinations.

In Fig. 2.3, the edges between a parent feature and a group of child features $\mathtt{f}_i$ are connected via an empty arc. This graphical element denotes a choice of exactly one feature out of a feature group (that is, choose one from $\{\mathtt{f}_1 \ldots \mathtt{f}_n\}$). In propositional logic, it is a generalization of an exclusive disjunction. Typical examples of exclusive disjunctions of features are different implementations of the same functionality or

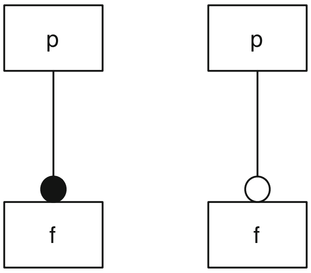


Fig. 2.2 Graphical notation for optional and mandatory features. A filled bullet denotes a mandatory feature, and an empty bullet denotes an optional feature

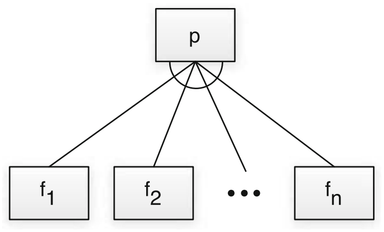


Fig. 2.3 Graphical notation for a one-out-of many choice. This choice corresponds to a generalized `xor` operator

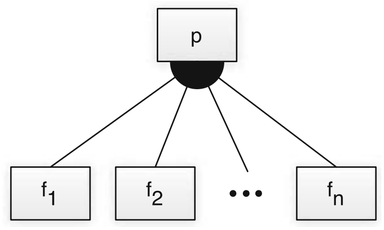


Fig. 2.4 Graphical notation for a some-out-of-many choice. This choice corresponds to the logical `or` operator

different technical platforms such as the choice of the supported operating system. This construct is called *alternative* or *mutually exclusive* choice.

Figure 2.4 shows child features connected via a filled arc. This graphical element denotes an unrestricted choice of one or more features out of a feature group. It is chosen if, at least, one feature of the collection has to be selected, but there are no other restrictions. Mathematically, it denotes an inclusive disjunction.

Example 2.9 Selecting one or several supported data types for storage is an example for an unrestricted choice in the domain of embedded data management. For the graph library, we may select one or more algorithms (any combination of algorithms is possible). □

The notational elements of feature diagrams support a natural description of a wide range of variability schemata, but not all. More general restrictions are needed in the form of propositional logic constraints. Typical constraints are implications between features located in different parts of the feature hierarchy, for example, to express that a certain algorithm requires a special data structure or that a certain function is not available for a certain operating system. Additional constraints can be simply added as arrows or in textual form to the diagram. Those additional constraints may span large parts of the feature diagrams and are therefore called *cross-tree constraints*.

There is no clear rule of when to use a hierarchical decomposition and when to use cross-tree constraints. In principle, all feature dependencies could be expressed as cross-tree constraints over features that are all marked as optional. Typically, a hierarchical decomposition is used to structure a maximal space of features, whereas cross-tree constraints are used sparingly for remaining constraints that do not fit into the chosen hierarchy. As usual in modeling, there is no single 'best' answer. We will see in Sect. 2.3.3 that there can be many equivalent answers.

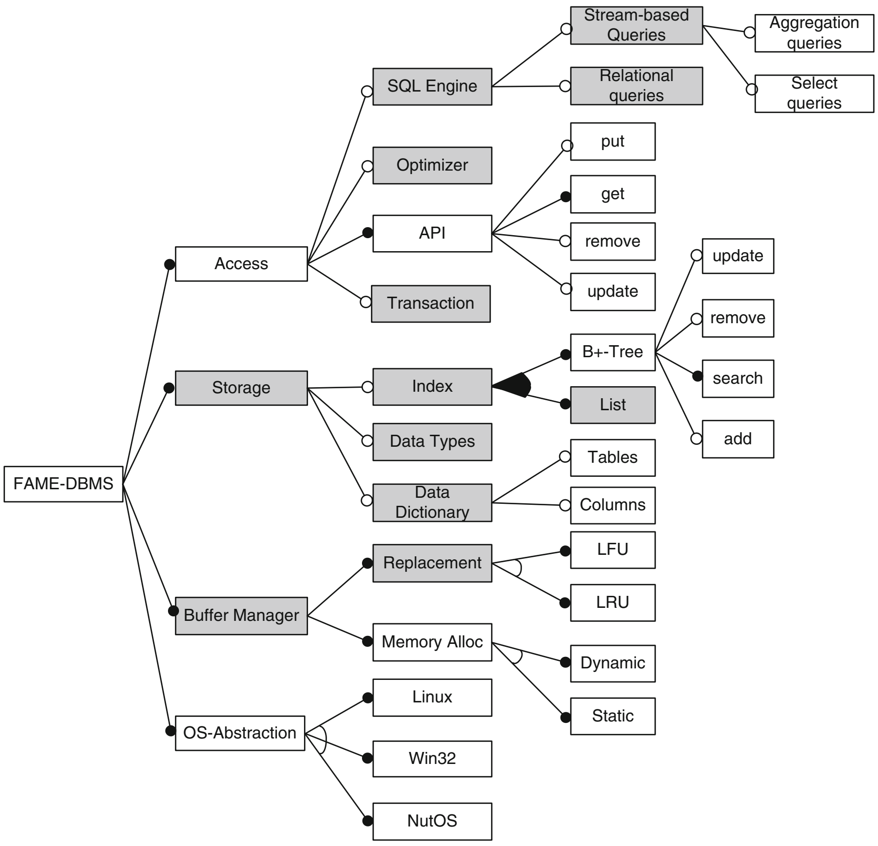


Fig. 2.5 Sample feature diagram for embedded data management

Example 2.10 For our embedded data management example, several partial feature diagrams are published (Rosenmüller et al. 2008; Rosenmüller et al. 2009b; Saake et al. 2009; Siegmund et al. 2009b). Figure 2.5 shows an excerpt of a feature diagram that, in its complete form, covers 65 features (Rosenmüller et al. 2011). Nodes shaded in gray are folded subtrees. □

Example 2.11 For embedded data management, storing an explicit data dictionary requires the support of the `String` data type to store attribute names:

$$\texttt{DataDictionary} \Rightarrow \texttt{String}$$

For the graph library, a typical cross-tree constraint would be that the computation of minimal spanning trees requires undirected, weighted edges:

$$\texttt{MST} \Rightarrow \texttt{Undirected} \wedge \texttt{Weighted}$$

□

For our presentation, we concentrate on Boolean features identified by a name. In principle, non-Boolean features or attributes of features may also be of interest in distinguishing products. For example, in a system supporting parallelization, the number of supported processors may lead to different products. Several dialects of feature models support non-Boolean features or non-Boolean attributes of features.

2.3.3 Formalization in Propositional Logic

Feature diagrams can be directly mapped to propositional formulas, thereby defining a formal semantics of feature diagrams. All feature names from the set `F` of feature names are interpreted as propositional variables. In the following, `p`, `f` and $\mathtt{f}_\mathtt{i}$ exemplify members of `F`.

A *mandatory* feature definition `mandatory(p,f)` between a parent feature `p` and a child feature `f` (denoted by a filled bullet at the child feature `f`) corresponds to a logical equivalence. That is, whenever the parent feature is selected, so too must the child and vice versa:

$$\mathtt{mandatory(p,f)} \equiv \mathtt{f} \Leftrightarrow \mathtt{p}$$

An *optional* feature, denoted by an empty bullet, is written as `optional(p,f)` and corresponds to implication. The implication states that the parent `p` may be chosen independently from `f`, but the child `f` can only be chosen if `p` is selected:

$$\mathtt{optional(p,f)} \equiv \mathtt{f} \Rightarrow \mathtt{p}$$

The *alternative* constraint defines a one-out-of-many choice and is denoted by an empty arc in feature diagrams. The definition $\mathtt{alternative}(\mathtt{p},\{\mathtt{f}_1,...,\mathtt{f}_n\})$ has as first parameter the parent feature `f` and as second parameter a non-empty set $\{\mathtt{f}_1,...,\mathtt{f}_n\}$ of child features. Mapped to propositional logic, this is a disjunction, in which, at least, one child feature is selected when the parent is chosen. Additionally, we ensure for each pair of child features that no two child features are selected together.

$$\mathtt{alternative}(\mathtt{p}, \{\mathtt{f}_1, ..., \mathtt{f}_n\}) \equiv ((\mathtt{f}_1 \vee \ldots \vee \mathtt{f}_n) \Leftrightarrow \mathtt{p}) \wedge \bigwedge_{\mathtt{i}<\mathtt{j}} \neg(\mathtt{f}_\mathtt{i} \wedge \mathtt{f}_\mathtt{j})$$

An unrestricted *choice* or *or*, denoted by a filled arc in feature diagrams, defines a some-out-of-many choice. Again, the definition $\mathtt{choice}(\mathtt{p},\{\mathtt{f}_1,...\mathtt{f}_n\})$ has as second parameter a non-empty set of child features. Mapped to propositional logic, the selection of `p` is equivalent to a disjunction of the child features.

$$\mathtt{or}(\mathtt{p}, \{\mathtt{f}_1, ...\mathtt{f}_n\}) \equiv (\mathtt{f}_1 \vee \ldots \vee \mathtt{f}_n) \Leftrightarrow \mathtt{p}$$

Additional cross-tree constraints can be expressed as propositional formulas, as well. In fact, they are often already specified as propositional formulas the corresponding graphical notations. Therefore, we can directly reuse them in the logic representation. All formulas for cross-tree constraints are connected via conjunction, which restricts the set of possible products.

Our formalization is summarized in Definition 2.4:

Definition 2.4 A *feature diagram* is a graphical representation of a feature model as a tree over the feature set `F`. Each edge in the tree is defined by exactly one feature constraint, that is, by a declaration of one of the feature constraint types *mandatory*, *optional*, *alternative*, or *or*.

$$\mathtt{root(f)} \equiv \mathtt{f} \tag{2.1}$$

$$\mathtt{mandatory(p,f)} \equiv \mathtt{f} \Leftrightarrow \mathtt{p} \tag{2.2}$$

$$\mathtt{optional(p,f)} \equiv \mathtt{f} \Rightarrow \mathtt{p} \tag{2.3}$$

$$\mathtt{alternative}(\mathtt{p}, \{\mathtt{f}_1, ...\mathtt{f}_n\}) \equiv ((\mathtt{f}_1 \vee \ldots \vee \mathtt{f}_n) \Leftrightarrow \mathtt{p}) \wedge \bigwedge_{\mathtt{i<j}} \neg(\mathtt{f}_\mathtt{i} \wedge \mathtt{f}_\mathtt{j}) \tag{2.4}$$

$$\mathtt{or}(\mathtt{p}, \{\mathtt{f}_1, ...\mathtt{f}_n\}) \equiv (\mathtt{f}_1 \vee \ldots \vee \mathtt{f}_n) \Leftrightarrow \mathtt{p} \tag{2.5}$$

Additionally, a set of *cross-tree* constraints may be defined. The corresponding propositional formula of the feature constraints and the cross-tree constraints are conjoined resulting in one propositional formula that represents the semantics of the whole feature diagram. □

Propositional logic enables us to use automated tools such as SAT solvers to test interesting properties, such as checking validity of feature models and feature selections, detecting dead features, and comparing feature models. In Chap. 10, we explore different use cases of automated analyses of feature models based on the mapping defined here.

2.3.4 *The Feature Model for the Graph Library*

We use the product line of graph libraries from Sect. 1.5 to illustrate feature diagrams and their formalization. Recall, this product line supports variations in a library of graph data structures and algorithms.

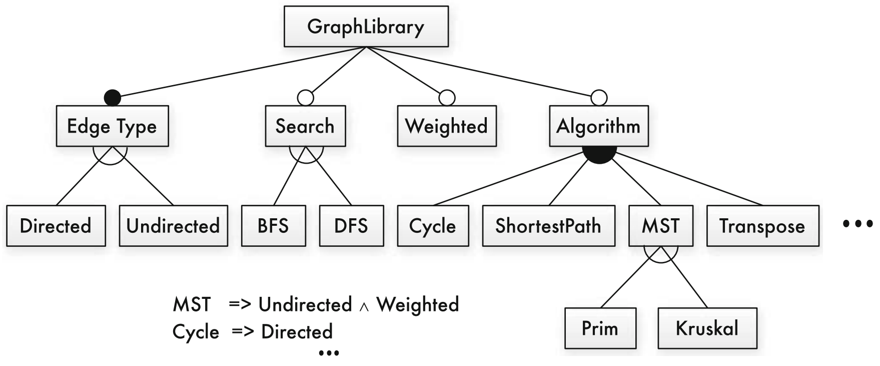


Fig. 2.6 A possible feature diagram of the graph library

A possible feature diagram for the graph library is shown in Fig. 2.6. The root is labeled with GraphLibrary to represent a graph product (that is, a graph library). It has a mandatory child feature EdgeType, because each graph library has to implement an edge type, which is either Directed or Undirected. Furthermore, three other child features of the root are optional: Search, Weighted, and Algorithm. Search strategies may be either breadth-first search (BFS) or depth-first search (DFS). Algorithm offers a selection of graph algorithms as child features. Since it is optional, either zero, one, or more algorithms may be present in a graph product. In our example, the algorithm for minimal spanning trees MST has two alternative implementations, Prim and Kruskal. Some non-local conditions are modeled as explicit Boolean constraints—for example, minimal spanning trees make only sense for weighted graphs, and shortest paths can be computed for directed graphs only.

Next, we use the feature diagram to illustrate the mapping to a propositional formula as introduced in Definition 2.4. The diagram is equivalent to the following conjunction:

$$
\begin{aligned}
& \texttt{root(GraphLibrary)} \\
\wedge\ & \texttt{mandatory(GraphLibrary,EdgeType)} \\
\wedge\ & \texttt{optional(GraphLibrary,Search)} \\
\wedge\ & \texttt{optional(GraphLibrary,Weighted)} \\
\wedge\ & \texttt{optional(GraphLibrary,Algorithm)} \\
\wedge\ & \texttt{alternative(EdgeType,\{Directed,Undirected\})} \\
\wedge\ & \texttt{or(Search,\{BFS,DFS\})} \\
\wedge\ & \texttt{or(Algorithm,\{Cycle,ShortestPath,MST,Transpose\})} \\
\wedge\ & \texttt{alternative(MST,\{Prim,Kruskal\})} \\
\wedge\ & (\texttt{MST} \Rightarrow \texttt{Weighted}) \\
\wedge\ & (\texttt{Cycle} \Rightarrow \texttt{Directed}) \\
\wedge\ & (\cdots)
\end{aligned}
$$

After expanding the feature constraints, we arrive at the following formula:

$$
\begin{aligned}
& \texttt{GraphLibrary} \\
\wedge\ & (\texttt{EdgeType} \Leftrightarrow \texttt{GraphLibrary}) \\
\wedge\ & (\texttt{Search} \Rightarrow \texttt{EdgeType}) \\
\wedge\ & (\texttt{Weighted} \Rightarrow \texttt{EdgeType}) \\
\wedge\ & (\texttt{Algorithm} \Rightarrow \texttt{EdgeType}) \\
\wedge\ & (((\texttt{Directed} \vee \texttt{Undirected}) \Leftrightarrow \texttt{EdgeType}) \wedge \neg(\texttt{Directed} \wedge \texttt{Undirected})) \\
\wedge\ & ((\texttt{BFS} \vee \texttt{DFS}) \Leftrightarrow \texttt{Search}) \\
\wedge\ & ((\texttt{Cycle} \vee \texttt{ShortestPath} \vee \texttt{MST} \vee \texttt{Transpose}) \Leftrightarrow \texttt{Algorithm}) \\
\wedge\ & (((\texttt{Prim} \vee \texttt{Kruskal}) \Leftrightarrow \texttt{MST}) \wedge \neg(\texttt{Prim} \wedge \texttt{Kruskal})) \\
\wedge\ & (\texttt{MST} \Rightarrow \texttt{Weighted}) \\
\wedge\ & (\texttt{Cycle} \Rightarrow \texttt{Directed}) \\
\wedge\ & (\cdots)
\end{aligned}
$$

Both the diagram and formula are incomplete where ellipses appear.

The graph example does not contain an alternative construct with more than two child features. To illustrate the transformation to propositional logic for such situations, we use an additional example. Assume that our product line runs on different operating systems. This is modeled by a feature OS with three different alternative child features Linux, Win and Mac. Figure 2.7 shows the corresponding subtree of the feature diagram. This leads to the feature constraint `alternative(OS, {Linux, Win, Mac})`, which translates into the following formula:

$$\big(\texttt{OS} \Leftrightarrow (\texttt{Linux} \vee \texttt{Win} \vee \texttt{Mac})\big) \wedge \big(\neg(\texttt{Linux} \wedge \texttt{Win}) \wedge \neg(\texttt{Linux} \wedge \texttt{Mac}) \wedge \neg(\texttt{Win} \wedge \texttt{Mac})\big)$$

The number of pairwise exclusions is quadratic in the number of alternative features (each combination of two alternative features forms a negated clause).

2.3.5 Variations and Extensions of Feature Models

Feature models are widely used in research and practice. However, no standardized modeling format has been accepted, so far. Standards are on the way—at the time of

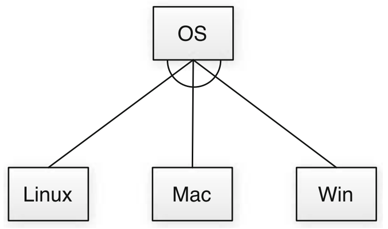


Fig. 2.7 Feature diagram for alternative operating systems

writing, there is a draft from the Object Management Group[1] and an Eclipse incubator project[2]—but they are not yet in a mature form. As a result, different notations and file formats are omnipresent.

In this book, we use a simple form of feature diagrams. Each pair of nodes participates in maximally one of the basic four feature constraint types; so, for example, it is not possible to add an optional feature as an edge in an alternative group.

In the literature, there are many variations of feature diagrams including:

1. Some cross-tree constraints can be modeled graphically. Arrows can denote implications or mutual exclusion, as exemplified in Fig. 2.8a.
2. Some notations distinguish *abstract* from concrete features. Abstract features are used for structuring and documentation purposes only and are not bound to implementation artifacts. They structure a diagram but do not reflect actual variability in the domain (such as features EdgeType and Search in Fig. 2.6). We exemplify the difference in Fig. 2.8b, in which abstract features are denoted by gray boxes.
3. Some notations support multiple group types under the same feature. For example, in Fig. 2.8c, features I, J, and K share the same parent even though they belong to different groups. In our notation, we can express such combinations only by introducing additional abstract features.

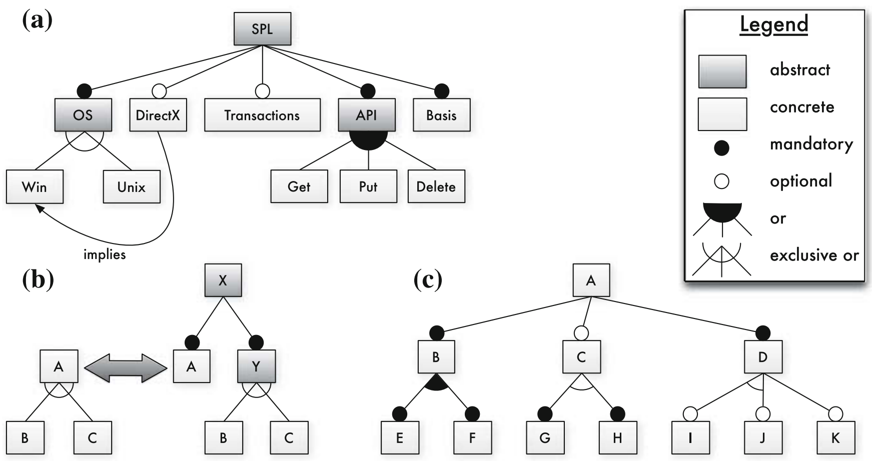


Fig. 2.8 Some variations of feature diagrams: **a** cross-tree constraints, **b** transformation toward abstract inner features, **c** mixing optionality and group constraints

[1] Common Variability Language (CVL), `http://www.omgwiki.org/variability`.

[2] EMF Feature Model, `http://www.eclipse.org/modeling/emft/featuremodel/`.

4. Some notations permit the mixing of mandatory and optional features with alternative and choice groups, as also illustrated in Fig. 2.8c. Czarnecki and Eisenecker (2000, Sect. 4.4.1.5) describe a normalization strategy for such models.
5. There are generalizations of or and alternative constructs, where a range or fixed number of options must be chosen (group cardinalities).
6. In some notations, a feature may be selected multiple times, each with different configurations of subfeatures (known as feature cloning or feature cardinalities).
7. In some approaches, features can have attributes. An algorithm of our graph product-line example may have the attribute memory footprint (indicating the amount of memory that is consumed by the feature) or a cost attribute (indicating the price of selecting the feature). Feature attributes may be useful in the automatic optimization of feature selections and product derivation.

Often, it is possible to convert between different formats, typically without loss of information, but possibly with loss of structure. Such details are beyond the scope of this book. The simple notation introduced in the previous sections is sufficient for our needs.

2.3.6 Feature Modeling in Practice

Sadly, research literature on feature models usually provides only small examples, along the lines of the graph library in Fig. 2.6. Although researchers have attempted to collect a corpus of example feature models,[3] feature models of industrial product lines are rarely publicly available.

In this section, we try to peek into industrial practice by looking at feature modeling in the Linux kernel, the largest real-world example of a publicly available software product lines, and by looking at a commercial product-line tool that scales to product lines with thousands of features. We briefly discuss their relation to the approach presented in this book.

Kconfig

The Linux kernel can be considered as a software product line. It has over 10,000 features, configurable at compile time. In version `2.6.28.6`, the feature model for the x86 architecture has 5,426 features, of which 4,744 are configurable by end users. Recently, its feature-modeling mechanism, the Kconfig language and tool, has been studied intensively (She et al. 2010; Berger et al. 2010b; Lotufo et al. 2010).

Instead of feature diagrams, the Linux developers use a self-written textual domain-specific language, Kconfig, to specify features and their dependencies. Kconfig closely resembles mechanisms of feature modeling, but uses a slightly different

[3] A large repository with over 200 feature models is available on the web: `http://www.splot-research.org/`.

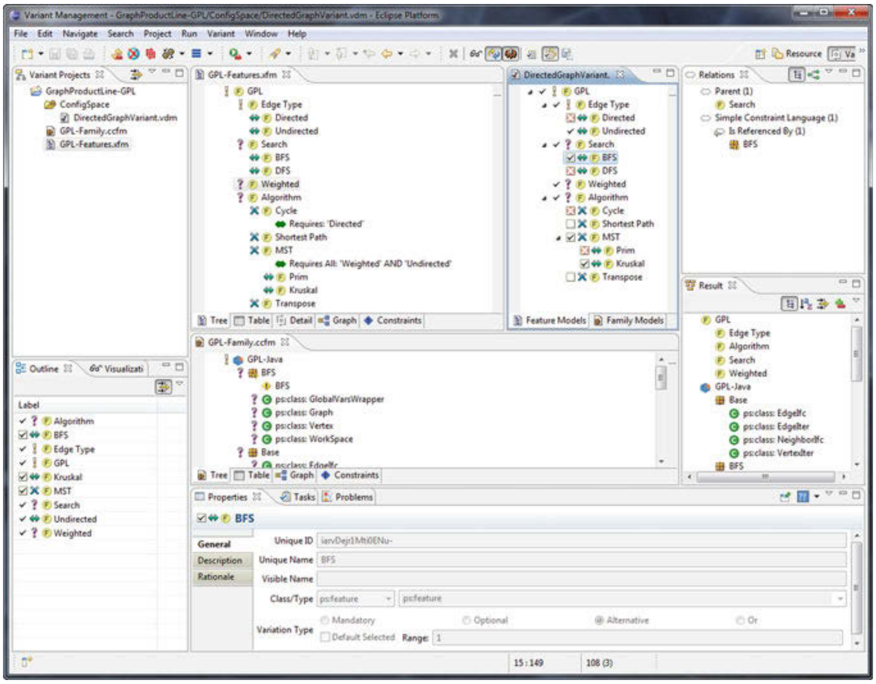
```
menu "Power management and ACPI options"
  depends on !X86_VOYAGER

  config PM
    bool "Power Management support"
    depends on !IA64_HP_SIM
    ---help---
      "Power Management" means that ...

  config PM_DEBUG
    bool "Power Management Debug Support"
    depends on PM

  config PM_STD_PARTITION
    string "Default resume partition"
    depends on HIBERNATION
    default ""
    ---help---
      The default resume partition is ...
endmenu
```

Fig. 2.9 Excerpt for the feature model of the Linux kernel, written in the Kconfig language (adopted from She et al. 2010)

terminology. In Fig. 2.9, we show a small excerpt of the feature model of the Linux kernel. It consists of a hierarchy of menus ('menu', roughly equivalent to abstract features, see Sect. 2.3.5) that contain configuration options ('config'). A configuration option can have different types, such as Boolean, tristate, integer, and string. Boolean configuration options are closest to our notion of features. Each configuration option has a name, a description, a type, and possibly defaults ('default'), and dependencies ('depends on'). Due to its size, the feature model of the Linux kernel is divided into multiple files (roughly corresponding to the overall architecture of Linux), with a lexical include mechanism to compose them together.

The Kconfig language comes with a set of tools that process a feature model and provide an interactive environment where a user can select desired features. During feature selection, a user explores a hierarchy of menus (entering submenus in *menu config* or using a tree structure in *xconfig*) and selects desired configuration options. Compared to many academic feature models, the menu structure of the Linux kernel is not deeply nested, but menus often have a large number of choices; there are many and complex cross-tree constraints between features, some involving up to 22 features. The tool infrastructure can hide features that are currently not selectable and can enforce constraints between users during configuration. It provides a rich environment to show help texts and dependencies. Still, in a user survey, Hubaux et al. (2012) have found that configuring the Linux kernel remains a challenging task.

Since the Linux kernel is open source, it has recently been the focus of many researchers interested in real-world feature models. For a closer look—including comparisons to other feature models, a discussion of problems, and a study on evolution of variability—see the recent research literature (She et al. 2010; Berger et al. 2010b; Lotufo et al. 2010; Hubaux et al. 2012).



BFS

Unique ID: iarvDejr1Mti0ENu-
Unique Name: BFS
Visible Name:
Class/Type: ps:feature ps:feature
Variation Type: Mandatory Optional [✓ Alternative] Or
Default Selected Range: 1

Fig. 2.10 Variability of the graph library modeled with pure::variants

pure::variants

There are a few commercial tool suites for the development of software product lines. Among them, *pure::variants*[4] has been used in many companies, often with large real-world feature models (Beuche et al. 2004).

Based on the Eclipse development environment, *pure::variants* provides facilities to model features in a hierarchical way. Instead of the textual notation of Kconfig and the graphical notation of feature diagrams, the primary editor of *pure::variants* is tree-based, in which developers add or change features using specific editor commands, as illustrated in Fig. 2.10. The editors in *pure::variants* use custom symbols, which can be mapped to the feature-diagram notation. Arbitrary cross-tree constraints can be added using additional relations, written in a Prolog dialect. In addition to traditional feature-modeling concepts, *pure::variants* provides constructs for soft constraints, similar to defaults in Kconfig.

Besides features names, *pure::variants* permits users to define additional descriptions (cf. Sect. 2.3.1) and feature attributes. Values of feature attributes can be predefined, precalculated, or set during configuration. Feature attributes can be used to restrict valid selections of features by defining constraints on them (cf. Sect. 2.3.5).

4 http://www.pure-systems.com.

For example, a constraint could specify "if the number of processors is larger than 3, select a different sorting algorithm." Attributes may also be provided during modeling, for example to specify the costs of a feature, which can then be shown during the configuration process.

Although *pure::variants* is not deployed with examples from industrial software product lines and published experience reports do not reveal many details, having a closer look at the tool can be instructive to see what kinds of mechanisms are needed for a feature-modeling tool in practice. A free community edition of the tool is available for experimentation. We return to *pure::variants* and its facilities beyond feature modeling in Appendix A.

2.3.7 Tooling

There are several tools that can be used to describe feature models. In the simplest case a simple text processor is sufficient to describe a list of features and their properties and relationships. More advanced tools that target feature models directly, can provide additional tool support, such as enforcing a consistent structure, dividing models into smaller modules and composing them, reasoning about features (see also Chap. 10), supporting the requirements-analysis process based on the feature model and much more.

Apart from commercial tools, over the years, many different academic feature-modeling tools have been created, such as Captain Feature[5] or the Feature Modeling Plug-in[6]—often in student projects, often now abandoned. One of the more stable academic tools is *FeatureIDE* (described in more detail in Appendix A), which contains a graphical feature-model editor conforming largely to the graphical feature-diagram notation used in this book.

In open-source systems, textual modeling notations are more common, such as *Kconfig* from the Linux kernel described above. The eCos operating system for embedded devices also comes with a powerful feature-modeling notation *CDL* and suitable tools. For more details on these open-source feature modeling tools, we recommend the survey of Berger et al. (2010b).

The two main commercial product-line tools *pure::variants* and *Gears* also support feature modeling, as described further in Appendix A.

2.4 Adoption Paths of the Product-Line Approach

How should one start the development of a software product line in the first place? In most cases, a company moving to product-line technology has already some products of the target domain in its portfolio. In these cases, we think of a *transition* to

[5] `http://captainfeature.sf.net/`

[6] `http://gsd.uwaterloo.ca/fmp`

a product line, rather than of a development from scratch. Following Krueger (2002), we distinguish three different adoption paths:

1. The *proactive approach* develops a product line from scratch by carefully using analysis and design methods.
2. The *extractive approach* starts with a collection of existing products and incrementally refactors them to form a product line.
3. The *reactive approach* begins with a small, easy to handle product line (possibly consisting only of a single product) and is extended incrementally with new features and implementation artifacts, thus extending the product line's scope.

Since the adoption path can have a significant effect on the selection of implementation methods, we will look at all three approaches in detail.

2.4.1 Proactive Approach

The *proactive approach* is to develop a product line from scratch. In a design process as outlined in the previous sections, developers model the domain and implement all relevant features before the first product is generated. Typical tasks in the proactive approach are:

- domain analysis and scoping as explained before,
- deciding about the product-line implementation approach,
- and implementing the entire product line.

Using the proactive approach, developers can plan the product line's variability perfectly for the desired variability. As a result, one can reach a high level of code quality and maintainability. However, its drawback is a high upfront investment and corresponding risks before the first product arrives at the market. Moreover, with existing products, essentially a company has to stop production for a significant period of time for restructuring or even rewriting the code.

The proactive approach, as a clean-slate approach that follows academic habits, is often taught in product-line texts, and also the steps outlined in Sect. 2.2 (p. 19) follow this idealized model. There are several success stories from companies adopting a product line with a proactive approach, including a full production stop (Clements and Krueger 2002). However, it is debatable how applicable this process is in general. Often, some products are already in productive use and a long delay to transit to product-line technology is not acceptable. The proactive approach is often seen as idealistic and academic, which, in practice, has to be combined partly with ideas from the other two adoption strategies.

2.4.2 Extractive Approach

The *extractive approach* is useful when a company already has a portfolio of related products that target a common domain, but those projects are not engineered in a systematic way yet. Often companies start with a clone-and-own approach, where variations of a system are created by copying the source code and modifying the copy (or by creating branches in a version control system, see Sect. 5.1, for an example) to satisfy the specific needs of a customer. The more copies need to be maintained and evolved separately, the more expensive maintenance tasks become and the more developers run into maintenance problems, such as inconsistent evolution of different copies. Typically, at some point, the pressure on developers grows so strong that they are forced to adopt a more disciplined product-line approach, in which variations and commonalities are planned and structured intentionally. The aim of the extractive approach is to make a transition from one or multiple legacy products to a more structured product line.

Typical tasks of the extractive approach are:

- identification of commonalities and differences of existing products, based on domain knowledge and stakeholder requirements,
- extraction or implementation of the core functionality in the form of common reusable domain artifacts,
- and extraction and realization of the variation using appropriate implementation techniques.

The extractive approach advocates an incremental adoption of product-line technology. Common parts are extracted, and some cloning is eliminated step by step. Due to its incremental nature, risks and upfront investment are much lower compared to the proactive approach. During the adoption process, all products remain in production. However, the quality of the extracted product line relies on the quality of the tools supporting the extraction. Development does not follow the clear stepwise academic process of domain analysis followed by domain implementation. Since the extracted code fragments do not follow preplanned guidelines, but rely on existing code, the resulting code basis may be hard to maintain. Hence, the extractive adoption path potentially limits the choices of implementation techniques, as discussed in Part II of this book. As a lightweight, low-risk strategy, however, the extractive adoption path is typical in practice.

2.4.3 Reactive Approach

The *reactive approach* is an instance of Boehm's *spiral model* (Boehm 1985), an agile method to adopt a product-line approach. Developers start with a software product line $\mathtt{SPL_0}$ which realizes an initial version of the envisioned software product line. In incremental steps from $\mathtt{SPL_i}$ to $\mathtt{SPL_{i+1}}$, the product line progressively grows toward its ideal, covering the full variation spectrum, as defined during domain analysis (which can also be incremental).

Typical tasks in the reactive approach are:

- exploration and characterization of the requirements leading to a new product currently not covered by the product line,

- describing the delta leading to the improved product,
- and implementing the delta in a suitable way.

Besides being an adoption path, the reactive approach describes also a typical pattern for maintaining and evolving a product line during its lifetime.

Conceptually, reactive adoption is positioned between the proactive and the extractive approach. It requires less upfront planning than the proactive approach, but including a feature may require invasive and expensive changes to the product line, because it has not been designed with that feature in mind. At the same time, the reactive approach is typically considered to be more structured than the extractive approach, because each iteration follows clear planning steps. Overall, the reactive process aligns well with agile methods of software construction.

2.5 Further Reading

Feature-oriented domain analysis is well-explored in literature. For interested readers, who would like to learn more about domain analysis, we recommend the material of Kang et al. (1990), Simos (1995), and Czarnecki and Eisenecker (2000).

Examples of dialects and extensions of feature models are common in the product-line literature, for example, by Griss et al. (1998), Streitferdt et al. (2003), Beuche et al. (2004), Czarnecki et al. (2005a), Schobbens et al. (2007) and Michel et al. (2011). A well-explored alternative to feature models are decision models (Schmid et al. 2011). Czarnecki et al. (2012) provide a detailed discussion of commonalities and differences, and compare both feature models and decision models to tools used in practice. Furthermore, Pohl et al. (2005) advocate an alternative variability-modeling notation—orthogonal variability models.

The translation of feature models to propositional formulas has been described in several publications (Batory 2005; van der Storm 2004; Schobbens et al. 2007). The reverse direction, that is, reverse engineering of feature models from propositional formulas, was explored by She et al. (2011). Benavides et al. (2010) discuss the automated analysis of feature models, which we will explore further in Chap. 10.

The different adoption paths for product lines have been discussed by Krueger (2002). Clements and Krueger (2002) had a public controversial discussion on the trade-offs between different adoption approaches.

Exercises

2.1. Define the terms product line, feature, feature selection, feature dependency, product, scoping, and domain and give an example each.

2.2. What are the possible motivations for adopting the idea of a mass customization and product lines from industrial manufacturing also for software? What are typical scenarios in which product-line technology was or can be adopted for software? Why not simply create an application that contains all possibly needed features?

2.3. Find a physical product or software system that can be ordered customized over the Internet (PCs, cars, cloth, food, and so forth) with at least 8 configuration options.

(a) Describe the configuration space of the product using a feature model. Pay attention to possible dependencies between features.
(b) Estimate the number of possible configurations.
(c) Discuss the economic benefits and challenges of product lines for the vendor. What difference does it make when customers can select a specific configuration instead buying the standard configuration or selecting from a small set of preconfigured products?

2.4. Imagine a company that provides tailor-made chat software for the intranet of large cooperations (similar to IRC, ICQ, or Skype).

(a) Analyze the domain. Which features are likely to be requested by many customers? Which features are likely to be requested only by few customers? Which features could distinguish your products from the products of your competitors in this market segment?
(b) What advantages does product-line technology provide in this context? Discuss also alternative solutions.
(c) Model the domain with a feature diagram. Pay attention to feature dependencies.
(d) Translate the feature diagram into a propositional formula.
(e) Name valid and invalid feature combinations with respect to the feature model.

2.5. Repeat Exercise 2.4 with different domains, for example,

(a) Webmail applications,
(b) embedded firmware for consumer television sets,
(c) operating systems for smartphones,
(d) word processors,
(e) control software for diesel engines,
(f) drivers for graphic chips,
(g) satellite navigation systems,
(h) firmware for printers, and
(i) development environments, such as Eclipse or Visual Studio.

2.6. Translate the feature model from Fig. 2.5 (p. 30) into a propositional formula.

2.7. Why is it useful or even necessary to limit the scope a product line to a specific domain? Discuss the scope of a potential software product line for multimedia systems and a potential software product line of drivers for a fingerprint scanner. What information would you need to make informed scoping decisions?

2.8. Explain the general process of developing a software product line. Explain the steps with the chat example from Exercise 2.4. What are the differences and commonalities between typical processes for developing a single application and the process of developing a software product line?

2.9. Explore pros and cons of the individual adoption paths discussed in Sect. 2.4 for the following scenarios:

(a) A small startup has developed a prototype of an innovative satellite navigation system with a revolutionary new routing algorithm. To release the software to a well-defined set of hardware and software platforms the team decides to pursue a product-line approach.
(b) Dozens of teams in the IT department of a large consumer electronics manufacturer have separately built individual software for different TV receivers for different markets world wide for many years. However, management becomes increasingly impatient with slow release cycles and wants to adopt a product-line approach.
(c) A producer of mobile phones tries to enter the market of low-energy solar-powered phones with limited functionality. Since the hardware is innovative and different, most software will have to be rewritten, but also several well-defined subsystems for voice processing might be reusable. As the market evolves quickly and unpredictably (it might be necessary to quickly copy a feature when a competitor innovates one), a few committed developers suggest developing the operating system as a product line.

Document what additional information you would need to make an informed decision about a suitable adoption strategy.

2.10. Discuss reasons why feature models are typically represented as trees and not as lists, graphs, logic expressions, scripts, or prolog programs. When feature models are represented as trees in feature diagrams, why are cross-tree constraints still needed? Discuss whether cross-tree constraints should be replaced with additional graphical notations.

2.11. Give an example of two feature diagrams representing the same set of valid products. Discuss whether there is a normal form or there should be one.

Part II
Variability Implementation

Chapter 3
Basic Concepts, Classification, and Quality Criteria

After reading the chapter, you should be able to

- characterize product-line implementation techniques using a suitable vocabulary (especially, binding times, language-based versus tool-based, and annotation versus composition), and
- discuss and compare product-line implementation techniques based on proper quality criteria (especially, preplanning effort, feature traceability, separation of concerns, information hiding, granularity, and uniformity).

In Part I, we described a process to develop feature-oriented product lines. It involves domain and application engineering, each comprising several phases, from domain and requirements analysis to implementation and product derivation. Since the early days of software product lines, the overall process and the analysis phase have been at the forefront of interest (problem space with domain and requirements analysis in Fig. 2.1). Researchers and practitioners were mostly concerned with identifying and specifying features and their relationships (see Sect. 2.3), exploring paths of product-line approaches for industrial adoption (see Sect. 2.4), as well as extending and adapting processes taking product-line methods into account (see Sect. 2.4). The fact that software product lines have to be implemented *systematically* and *efficiently* in order to attain the ambitious goals of facilitating reuse, variation, and automated software construction (solution space with domain implementation and product derivation in Fig. 2.1) was not the center of interest.

In research and practice, a multitude of mechanisms, languages, and tools have been developed that strive for supporting the development of reusable and variable software. Mostly, they were not specific to software product lines nor did they take the specifics of the product-line process into account. Nevertheless, several have been adopted silently by product-line developers, while others are on their way into practice. In Chaps. 4 and 5, we introduce and discuss techniques that are already standard in the practice of product-line engineering and that have been invented largely without software product lines in mind. In Chaps. 6 and 7, we review recent approaches that are in early stages of being transferred into practice, mostly tailored to the specific requirements of software product lines.

In this chapter, we introduce basic concepts that are central to product-line implementation techniques and that we use in the remaining chapters. In particular, we introduce three dimensions for classification in Sect. 3.1 as well as six quality criteria for comparison of product-line implementation techniques in Sect. 3.2.

3.1 Dimensions of Variability Implementation

To implement software product lines efficiently, the underlying code has to be *variable*. As variability is a very general concept (Svahnberg et al. 2005), we use the following definition throughout the book that is tailored to product-line engineering:

Definition 3.1 *Variability* is the ability to derive different products from a common set of artifacts. □

The desire for variable software is driven by the typically broad and diverse spectrum of requirements of the stakeholders of a product line, manifested in features they want in a product. Instead of serving the needs of a particular stakeholder by developing a product from scratch, variable software can be *tailored* stakeholder requirements. There is a multitude of techniques developed for this purpose. In the remaining section, we introduce three dimensions for classifying variability implementation techniques.

3.1.1 Binding Time

Variability offers choices. When we derive a product, we make decisions; we decide which features will be included in the product or not. We also say that we *bind* a decision. Different implementation techniques allow binding decisions at different times. In other words, they allow different *binding times*.

We distinguish between compile-time binding (also called early binding, static binding, or static variability), load-time binding, and run-time binding (also called late binding, dynamic binding, and dynamic variability).[1] With an implementation technique that supports compile-time binding, developers make decisions of which features to include at or before compile time. Code of deselected features is then not even compiled into the product. Examples include implementing variable code

[1] Some researchers distinguish between even more binding times, including preprocessing-time, link-time, weaving-time binding, and so forth (Rosenmüller 2011). For our discussions, the three-level distinction into compile time, load time, and run time is sufficient.

with preprocessors (Sect. 5.3) and feature-oriented programming (Sect. 6.1). With an implementation technique that enables load-time binding, developers can defer feature selection until the program is actually started. That is, during compilation all variations are still available; they are decided after deployment, for example, through command-line parameters or configuration files. Some techniques even support run-time binding, where decisions are deferred to run time and may even change during program execution (dynamic reconfiguration triggered by external or internal stimuli). Examples of implementation techniques that support load-time and run-time variability include simple parameter-based variability (Sect. 4.1) and context-oriented programming (Sect. 6.6.3). Some implementation mechanisms support multiple binding times.

Definition 3.2 *Compile-time variability* is decided before or at compile time. *Load-time variability* is decided after compilation when the program is started. With run-time variability, decisions can be made and changed during program execution. □

The different binding times each have advantages and disadvantages. Typically, compile-time binding leaves room for more optimizations. All unnecessary code can be removed from the product, reducing run-time overhead in terms of binary footprint, memory consumption, and execution time. However, once the product has been generated and deployed, it is not variable any more—an issue that has been explored in depths in autonomic computing (Cheng et al. 2009).

A product with load-time variability is more flexible to reconfigure. Instead of recompiling a new product after each change, end users can modify parameters and restart the same binary product. A product in which variability decisions are even delayed to run time is able to react to internal and external stimuli (in our context, a feature request) by adapting its behavior. In product-line terminology, this behavioral adaptation at run time can be described as the derivation of *virtual* products. However, mechanisms for load-time and run-time binding often incur a memory and performance overhead, since all variations are compiled into a single binary and consistency conditions must be checked at run time. Including unnecessary functionality in a shipped product may also be considered as potential security threat. We discuss potential problems in more detail in the context of parameter-based implementations in Sect. 4.1.1 (p. 66).

3.1.2 Technology: Language-Based Versus Tool-Based

Another dimension of classifying different product-line implementation techniques is to distinguish those that are based on mechanisms provided by a programming language, called *language-based approaches*, from those that are based on tools that operate on software artifacts to derive products, called *tool-based approaches*. A classic language-based approach is to realize variability with run-time parameters (see Sect. 4.1). A classic tool-based approach is to use a preprocessor that transforms software artifacts based on a given feature selection (see Sect. 5.3). Unfortunately, one cannot always draw a sharp line between language-based and tool-based approaches (for example, some component approaches are language-based and some are tool-based), but this rough distinction is sufficient to structure our discussion.

Definition 3.3 A *language-based approach* uses the mechanisms provided by a host programming language to implement features and to derive products. A *tool-based approach* uses one or more external tools to implement or represent features in code and to control the product-derivation process. □

In a language-based approach, both the implementation of features as well as the feature and variability management are located in the source code. This makes it easy for developers and analysis tools to understand and reason about the product line and its implementation as a whole. However, depending on the implementation approach, feature boundaries and feature management tend to vanish in the code, for example, when using run-time parameters to control feature-specific behavior (see Sect. 4.1).

A tool-based approach favors a clear separation between feature implementation, on the one hand, and feature and variability management and product derivation, on the other. While this separation can simplify the code structure, it makes it necessary for developers and tools to locate, understand, and reason about multiple artifacts in different places.

3.1.3 Representation: Annotation Versus Composition

A major goal of feature-oriented product line engineering is to derive a product automatically from variable code, based on a user's feature selection. Hence, product derivation involves product generation, statically or dynamically. In this book, we concentrate on two approaches that are widely used in practice: *annotation-based* and *composition-based approaches*, which differ in the way they represent variability in the code base and the way they generate products.

In annotation-based approaches, the code of all features is merged in a single code base, and annotations mark which code belongs to which feature. In some sense, an annotation is a function that maps a program element to the feature or feature combination it belongs to. A standard example is implementing variable code using the C preprocessor (Sect. 5.3). The C code comprises the code of all products, and preprocessor directives within the C code control which code fragments are included or excluded upon a feature selection (by setting preprocessor constants). Also the use of parameters that control and alter the behavior of a program at run time can be viewed as an annotation. In this case, an `if` statement that executes code depending on the feature selection plays the role of an annotation.

Definition 3.4 *Annotation-based approaches* annotate a common code base, such that code that belongs to a certain feature is marked accordingly. During product derivation, all code that belongs to deselected features or invalid feature combinations is removed (at compile time) or ignored (at run time) to form the final product. □

Annotation-based approaches are widely used in practice because they are easy to use and already natively supported by many programming environments. Nevertheless, preprocessor-based and parameter-based implementations are often criticized for their potential complexity, lack of modularity, and reduced readability, as we discuss in Sects. 4.1 and 5.3. However, some disadvantages, such as scattered code or error potential can be addressed with relatively simple tool support, as shown in Chap. 7.

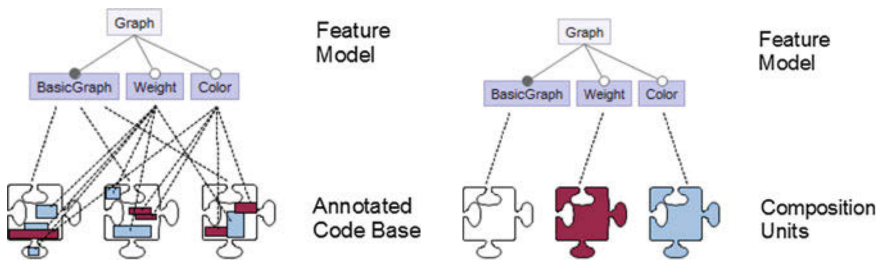
Composition-based approaches locate code belonging to a feature or feature combination in a dedicated file, container, or module. A classic example is a framework that can be extended with plug-ins, ideally one plug-in per feature; different products can be generated by integrating different plug-ins, as we discuss in Sect. 4.3.3. Beyond classic composition-based approaches, such as frameworks and components, there is a large body of research on advanced language abstractions and composition mechanisms to implement feature-oriented product lines. In Chap. 6, we discuss a number of such approaches including feature-oriented and aspect-oriented programming.

Definition 3.5 *Composition-based approaches* implement features in the form of composable units, ideally one unit per feature. During product derivation, all units of all selected features and valid feature combinations are composed to form the final product. □

The key challenge of composition-based approaches is to keep the mapping between features (problem space) and composition units (solution space) simple and tractable, ideally one-to-one (see Chap. 2). If each feature has its own implementation in the form of a composition unit, a generator can simply include the corresponding unit in the composition when a feature is selected.

In principle, any combination of annotation-based and composition-based approaches is possible (Kästner et al. 2009a). For example, we could decompose a system into composable units, where certain components are themselves variable in the sense that their implementations are annotated. During product derivation, a generator would select a subset of composition units and remove annotated code from them that belongs to deselected features.

Another way to view the difference between annotation and composition is that annotation-based approaches support *negative variability* (code is removed on



(a) Annotation-based approach **(b)** Composition-based approach

Fig. 3.1 Annotation-based and composition-based approaches to product-line implementation

demand), and composition-based approaches support *positive variability* (composition units are added on demand), or that annotation-based approaches *separate concerns virtually* and composition-based approaches *separate concerns physically* (see Chap. 7 for a more detailed discussion). Finally, we would like to emphasize that annotation and composition are two special but important instances of the broader set of possible program-generation and program-transformation mechanisms that can be used for product-line development. We illustrate the principle differences between annotation-based and composition-based approaches to product-line implementation in Fig. 3.1.

3.2 Quality Criteria

A key objective of this book is to convey that different implementation techniques for product-line development have different characteristics and mutual strengths and weaknesses. To assess tradeoffs and compare implementation strategies, we introduce and discuss six quality criteria that product-line implementation techniques should ideally meet: Low preplanning effort, feature traceability, separation of concerns, information hiding, granularity, and uniformity. As we will explain, not all quality criteria may be met at the same time, as some quality criteria pursue conflicting goals. Hence, different implementation strategies focus on different criteria and make different tradeoffs.

3.2.1 Preplanning Effort

Independently of the adoption path, product-line engineering always incurs a certain amount of *preplanning* (of course, more in the proactive approach than in the reactive approach, but still). Which features will be requested? Which features are likely to interact? Where will one feature extend the implementation of another feature?

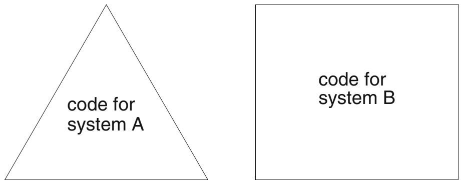


Fig. 3.2 Designs of two products of the same domain

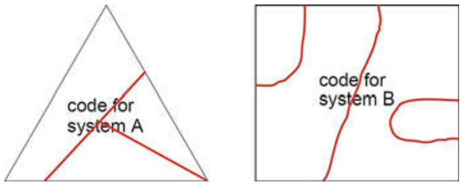


Fig. 3.3 Ad hoc decomposition of both products of Fig. 3.2

The key goal of preplanning is to ease the anticipation of changes and sources of variability and reuse. Different implementation techniques perform differently in facilitating or hindering this task. That is, the effort of preplanning an implementation technique requires is an important quality criterion.

By studying a representative set of software products of a domain, a set of design and implementation patterns will emerge. For example, consider the pair of shapes in Fig. 3.2, which represent the design and implementation of two products, A and B, that belong to the same domain D. Now we partition A and B into parts. Figure 3.3 shows a typical partitioning that we have experienced from students and engineers. Sometimes there are regular shapes, other times there are rough shapes.

Now, suppose product C is a natural addition to domain D (Fig. 3.4). Given the decomposition of Fig. 3.3, how easy is it to construct C? That is, can we reuse parts from other products? The answer is no. There is nothing "natural" about the shapes identified in Fig. 3.3 that would lend itself to the easy creation of C. With proper preplanning, there is a solution that allows all systems, A, B, and C, to be reconstructed in no time from existing parts, as illustrated in Fig. 3.5.

The whole product-line development process (see Sect. 2.2, p. 19) involves a form of preplanning. Instead of developing individual software systems, product developers analyze an entire domain and anticipate potential requirements in terms of features. The information about anticipated variations is a prerequisite for a proper design of a product line. However, not all features and variations can be anticipated, especially, in reactive and extractive adoption paths, see Sect. 2.4, p. 39.

Ideally, a product-line implementation technique minimizes the necessary preplanning effort and even allows implementing features that were not planned upfront with low effort. An important observation is that some design and implementation techniques foster change, such that preplanning is necessary only to a minor extent, while other approaches require substantial preplanning activities. For example, using frameworks (Sect. 4.3), a programmer has to anticipate where a given program may be extended by features in the future. Advanced implementation techniques such as feature-oriented programming (Sect. 6.1) and aspect-oriented programming

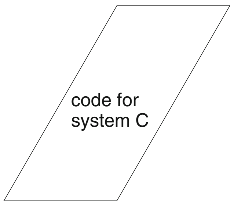


Fig. 3.4 Outline of the code base of another system in the same domain

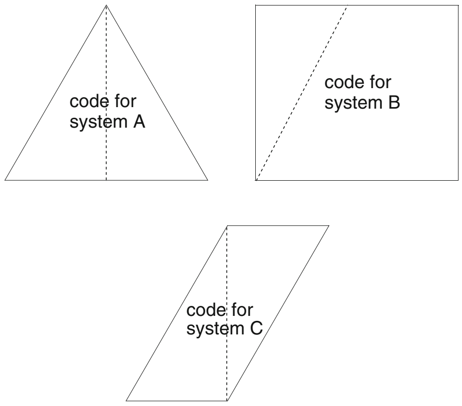


Fig. 3.5 Reusable decompositions of the products A and B

(Sect. 6.2) aim at reducing the need and effort for preplanning. For example, an aspect can extend an existing class by new members without changing the class. An extreme case, in which any extension can be made without preplanning, is discussed when introducing the concept of *obliviousness* in Sect. 6.2.4. Overall, fostering change and little preplanning effort is an important quality criterion for product-line implementation techniques.

3.2.2 Feature Traceability

In Fig. 3.1, it is apparent that the whole idea of feature-orientation and feature-based product derivation depends on establishing and managing the mapping between the problem and the solution space, in our case, between features and their implementation artifacts.

Definition 3.6 *Feature traceability* is the ability to trace a feature from the problem space (for example, the feature model) to the solution space (that is, its manifestation in design and code artifacts). □

Tracing features in design and code is a key property of product-line implementation techniques. Different techniques provide different levels of support for traceability. For example, while preprocessor directives are easily recognizable by tools and programmers (see Sect. 5.3, p. 110), run-time parameters are harder to trace, because they are difficult to distinguish from other variables and can be reassigned during execution (see Sect. 4.1, p. 66). In composition-based approaches, feature traceability is trivial as long as there is one composition unit per feature (see Chap. 6). If feature code is not properly separated in terms of dedicated units (for example, one feature is implemented as part of many components of a system), feature traceability is impaired.

3.2.3 Separation of Concerns

A fundamental principle in software design is to *separate concerns* (Parnas 1972; Dijkstra 1976). As defined in Sect. 2.1, a concern is an area of interest or focus in a system, and features are the concerns of primary interest in product-line engineering. A common approach to attain traceability (especially, in composition-based approaches) is to separate features both in design and code, such that the relationship between features and corresponding design and implementation artifacts are explicit.

When separating features into distinct artifacts, developers can easily find all code related to that feature for maintenance or evolution tasks. Related pieces of code are implemented together, which is known as *cohesion*. Cohesive pieces of code are typically easier to reason about than widely scattered code fragments.

In the history of programming languages and software engineering, a multitude of mechanisms to separate concerns have been developed, most notably procedures, modules, and classes. All of them facilitate one or the other form of hierarchical structure or block structure. In the 1990s, an insight emerged that a certain class of concerns, called *crosscutting concerns*, is inherently difficult to separate using these traditional mechanisms based on block or hierarchical structure (Kiczales et al. 1997).

Definition 3.7 *Crosscutting* is a structural relationship between the representations of two concerns. It is an alternative to hierarchical and block structure. □

Classic programming languages suffer from a limitation that is referred to as the *tyranny of the dominant decomposition*, which is the cause of crosscutting (Tarr et al. 1999): Using hierarchical structures, a program can be decomposed in only one way (along one dimension) at a time, called the dominant decomposition. All concerns that do not align with the dominant decomposition end up in *scattered* and *tangled code*. As developers, we have the freedom to choose how to decompose our system and which concerns to separate. However, we cannot separate all concerns at the same time; there will always be some concerns that do not align with the chosen dominant decomposition and that will be left scattered. In practice, this means that scattering and tangling is not necessarily a sign of bad design, but simply unavoidable.

Definition 3.8 *Code scattering* refers to a concern representation that is scattered across representations of multiple other concerns. □

Code scattering is related to code tangling.

Definition 3.9 *Code tangling* refers to the intermingled representation of several concerns within a module. □

In Fig. 3.6, we illustrate the effect of crosscutting by means of an example. The figure shows a snapshot of the code base of the Apache Tomcat server (in particular, the session subsystem). The code that belongs to the session-expiration concern is highlighted in red. This figure makes it clear that the code of this concern is scattered across the entire code base and tangled with the code of other concerns, not related to session expiration.

As features are often (but not necessarily) crosscutting concerns (Czarnecki and Eisenecker 2000; Apel et al. 2008b; Liebig et al. 2010), the ability to separate even crosscutting concerns into cohesive implementations is an important quality criterion of product-line implementation techniques.

3.2.4 Information Hiding

Separation of concerns aims to decompose a system into semantically cohesive parts. However, separation alone might not be sufficient if, to understand the whole, developers need to understand all details of all parts.

Information hiding is an essential concept in software engineering. The key idea is to decompose a system into modules (or components, see also Sect. 4.4), and to divide each module into an internal and an external part. The internal part is also known as the module's secret that is hidden (or encapsulated) from other modules and typically represents the bulk of the module's code, whereas the external part describes a contract to the rest of the system and is known as an interface. Ideally, with information hiding, each module can be understood in isolation by looking just at the module's secret and the interfaces of imported modules,[2] but not the secrets

[2] There are many different flavors of module systems, in which imports are expressed in different ways. The details are not relevant here. The interested reader may investigate and compare module systems for ML (Blume and Appel 1999) and Java (Gosling et al. 2005; Ancona and Zucca 2001) and Cardelli's calculus (Cardelli 1997).

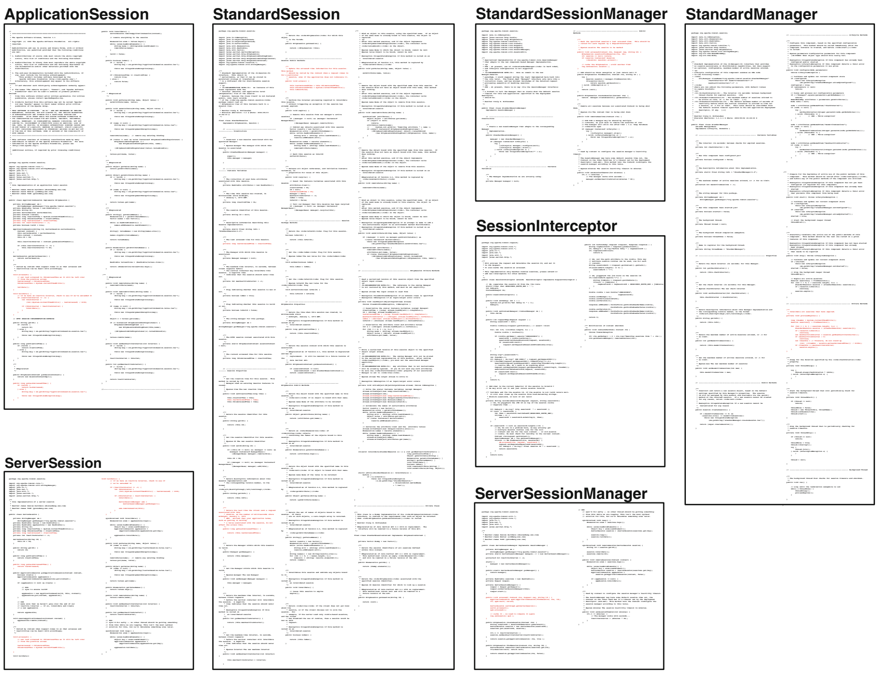


Fig. 3.6 Code of the session subsystem of the Apache Tomcat server; code responsible for session expiration is highlighted (Colyer et al. 2005)

of other modules—known also as *modular reasoning*. This way, we can be sure that the behavior of a module is not influenced by secrets of other modules.

Definition 3.10 *Information hiding* is the separation of a module into internal and external part. The internal part remains hidden from other modules, whereas the external part, the module's *interface*, specifies the contract of how the module interacts with the rest of the system. Information hiding enables *modular reasoning*, which means that developers can reason about a module without knowing the internals of other modules. □

There are many different ways to specify interfaces and how to hide module internals. All share that interfaces describe invariants between modules, but there are different views on how those invariants should be enforced. Interfaces can range from manually-enforced textual documentation to machine-checkable specifications of types, protocols, and so forth. Programming-language research constantly pushes the boundaries of what can be enforced mechanically, but textual specifications and documentation still remain just as important.

Information hiding has several important consequences. By reasoning about modules separately, we can divide work and assign it to independent teams (Parnas 1972). Developers do not need to inspect the entire code base when trying to understand an individual module. Developers may also change the internal implementation of a module freely as long as they preserve the behavior specified in the interface. The interface specifies the communication between modules and all communication is explicit in the interface. At a larger scale, we can understand the system in terms of module interfaces and compose modules by reasoning only about interfaces. Often, we can even compile and deploy modules separately. The key challenges of information hiding are to design small and clear interfaces to make communication explicit but also to maximize the secrets to hide.

When separating features, we would ideally also hide the internals of their implementation and make all communication between them explicit in interfaces. This frees developers from the burden of having to know all features in a system and how they may affect each other. Different teams can be responsible for different features and work in isolation on the basis of agreed interfaces. Therefore, information hiding is a quality criterion we strive for in many product-line implementation mechanisms. We will see that some implementation approaches (see *frameworks* and *components* in Sects. 4.3 and 4.4, pp. 79 and 89) use classic mechanisms with a long tradition of information hiding, whereas especially more recent approaches (for example, see *feature-oriented programming* and *virtual separation of concerns* in Sects. 6.1 and 7.4, pp. 130 and 184) separate concerns, but do not always provide suitable interface mechanisms, yet.

Without information hiding, separating concerns into cohesive artifacts may ease code navigation and reasoning, compared to scattered and tangled code, but it cannot give guarantees; in the worst case, developers need to know the entire system, as every module could potentially influence every other. On the other hand, information hiding without an educated decision of what to hide—which is typically driven by separation of concerns and planning for change (Parnas 1972, 1979)—may not separate the cohesive parts we want to reason about. For true modularity, we aspire to have both, separation of concerns and information hiding. Achieving true modularity typically requires significant preplanning though.

Note, in this book, we avoid the term *modularity*. Modularity is a highly overloaded term, on which many people project different ideas. For some, textually separating concerns into distinct files amounts to modularity. In contrast, for others, enforced information hiding with explicit and mechanically checked interfaces is essential. We use *separation of concerns* or *cohesion* when discussing separating implementations of feature into distinct artifacts and *information hiding* when discussing interfaces. We cannot avoid the term *module*, but we use it in the very broad sense of some file or container and discuss information hiding explicitly where appropriate.

3.2.5 Granularity

Features are rarely implemented in isolation from other features. A feature's utility arises from its connections with other features in the system. Only their collaboration gives rise to desired system behavior. To interact, a feature needs to induce changes, for example, by intercepting an event, registering a callback function, or injecting statements into an existing function.

Depending on the implementation technique, a feature can change a program at different levels of granularity. A *level of granularity* refers here to the hierarchical structure of an implementation artifact, typically, defined by a containment relation among the artifacts' structural elements, given by its syntax. Changes at the top of the hierarchical structure of an implementation artifact are *coarse-grained*. Changes at lower levels are *fine-grained*.

Example 3.1 A change induced by a feature that involves the addition of a new file or Java class to a given program is coarse-grained. Adding a new member to a given Java class is medium-grained, and adding a new statement to a given method body is fine-grained. □

Experience has shown that feature implementations take place at all levels of granularity (Kästner et al. 2008a; Liebig et al. 2010). Encoding fine-grained extensions in product-line implementation techniques that offer only mechanisms for coarse- or medium-grained extensions can be awkward and work-intensive. Usually, even fine-grained changes can be made with coarse-grained extensions, but at cost, for example, the cost of code replication, when replacing entire files with locally modified variants (we discuss an instance of this in the context of build systems in Sect. 5.2). If fine-grained extension points are known upfront, such as the need to inject statements inside a method, developers can often also extract those fine-grained locations to more coarse-grained structures, for example, by refactoring the statements to their own method or class.

The rule of thumb is that annotation-based approaches support more fine-grained changes and interactions than composition-based approaches, because the code of different features is intermixed anyway in a common code base, and annotations can be applied at nearly every place in a code base. Composition-base approaches rely on interfaces that are defined typically at the level of classes and methods.

3.2.6 Uniformity

Many concepts and mechanisms described in this book are independent of a particular implementation language, but for didactic reasons we use mostly Java as our host language. However, many product-line implementation techniques generalize too many languages. For example, the idea of intermixing code of different features in a single code base and to use annotations to establish a mapping between features and code (see *preprocessors* in Sect. 5.3, p. 110) is applicable to Java as well as C, C#, and any other textual language. Also, more sophisticated implementation mechanisms such as inheritance, component composition, or aspect weaving are not specific to certain languages (see Chaps. 4 and 6).

Also, in the context of software product lines, researchers began to view and develop their implementation approaches in a language-independent manner and began to develop language-independent tools and theories (Batory et al. 2004; Apel et al. 2009). Batory first formulated the principle of uniformity (Batory et al. 2004).

Definition 3.11 *Principle of Uniformity*: Features are implemented by a diverse selection of software artifacts and any kind of software artifact can be subject of subsequent changes and extensions. Conceptually, all artifacts (annotated or composed) should be encoded and synthesized in a similar manner. □

The principle of uniformity is not motivated by the fact that individual implementation mechanisms such as inheritance can be implemented for various languages, but by the observation that today's software systems, including software product lines, consist of many different kinds of software artifacts. Artifacts include *code* written in different languages as well as *noncode* artifacts, such as requirements, documentation, architecture descriptions, design and performance models, configuration files, and many more. Instead of inventing ad hoc approaches again and again for specific languages, which simply does not scale, it is desirable to have a general approach to be instantiated for all kinds of artifacts because there is only concept to remember. It is even possible, based on such a general approach, to develop largely language-independent tools for feature implementation and product derivation (see Sect. 6.1, p. 130).

The bottom line is that the principle of uniformity denotes an important quality criterion for product-line implementation techniques. Ideally, an implementation technique should be applicable, in principle and in practice, to a wide variety of different code and noncode artifacts.

3.3 Structure of Subsequent Chapters

In the next four chapters, we introduce, discuss, and compare different strategies to implement feature-oriented software product lines. In Table 3.1, we classify them according to the dimensions we introduced in Sect. 3.1. Some approaches have multiple instances with different properties, such that the classifications are not disjoint. For example, some component systems support both static composition and dynamic composition; others are both language-based and tool-based.

In addition, to the dimensions of Table 3.1, we distinguish between classic and advanced approaches: classic approaches that are used widely in practice and

Table 3.1 Classification of implementation approaches discussed in this book

	Binding time		Technology		Representation	
	Compile time	Load time (and later)	Language-based	Tool-based	Annotation	Composition
Parameters		✓	✓		✓	
Design patterns	(✓)[1]	✓	✓			✓
Frameworks	(✓)[1]	✓	✓			✓
Components	✓	✓	✓	✓		✓
Version control	✓			✓		✓
Build systems	✓			✓	(✓)[3]	✓
Preprocessors	✓			✓	✓	
Feature-oriented programming	✓	(✓)[2]	✓			✓
Aspect-oriented programming	✓	(✓)[2]	✓			✓
Virtual separation of concerns	✓			✓	✓	

[1] Both design patterns and frameworks support compile-time and run-time variability, but, in the case of compile-time variability, they induce still a run-time overhead. [2] Both feature-oriented and aspect-oriented programming support dynamic feature binding, but corresponding tools are rather experimental. [3] One can annotate an entire file to assign it to a particular feature

advanced approaches that are explored mostly in academia and that introduce novel tool-based or language-based implementation mechanisms. The chapter structure is guided by this distinction between classic and advanced mechanisms and the distinction into language-based and tool-based approaches. We choose this structure, because all strategies that we cover can be equally-balanced into language-based and tool-based—two philosophies largely independently explored, with mutual strengths and weaknesses. The other dimensions of classification are not explicit in the chapter structure of this book, but they still guide our discussions.

3.4 Further Reading

The concepts and classification introduced in this chapter are based on a wide variety of books and research papers. We provide selected pointers to further relevant original sources and text books. Czarnecki and Eisenecker discuss many of the concepts presented here more or less explicitly in their text (Czarnecki and Eisenecker 2000). Separation of concerns and especially crosscutting are discussed extensively in the literature on aspect-oriented programming (Filman et al. 2005; Laddad 2003). The distinction between annotation-based and composition-based approaches has been first proposed by Kästner and Apel (2008a; 2009a). The same authors discussed the role of granularity in product-line implementation (Kästner et al. 2008a). There are many different notions and interpretations of modularity, separation of concerns, and information hiding. The key ideas to hide information based on what is most likely to change and to build interfaces around the stable parts were explored in depth by Parnas (1972, 1976, 1979). We wrote down a detailed version of our current understanding of different notions of modularity, which provides a more in-depth discussion and a starting point for further exploration of that field (Kästner et al. 2011). The principle of uniformity was formulated by Batory et al. (2004). Svahnberg et al. present a taxonomy of variability-realization techniques, which is alternative to our classification (Svahnberg et al. 2005). Further classifications and surveys are available elsewhere (Anastasopoules and Gacek 2001; Muthig and Patzke 2002; Mezini and Ostermann 2004; Lopez-Herrejon et al. 2005).

Exercises

3.1. Discuss the strengths and weaknesses of product-line implementation approaches regarding:

(a) binding times (compile-time, load-time, and run-time variability),
(b) technology (language-based versus tool-based mechanisms), and
(c) variability representations (annotation versus composition).

3.2. Discuss which binding times are suitable (or ideal, or necessary) for the following features:

(a) Multiple alternative localization features (languages, metric versus imperial units, and so forth) for the graphical user interface of a satellite navigation system.
(b) Two alternative sorting features in a database system: a very fast and a power-efficient sorting algorithm.
(c) Two alternative features in an operating system: single-processor support and multi-processor support.
(d) Two alternative features edge representations in a library of graph algorithms: directed and undirected.
(e) Multiple optional features for supported file formats in printer firmware.

Which additional context knowledge would change your answer?
3.3. Is it possible to combine composition-based and annotation-based techniques in one approach? Would it be useful? Name possible scenarios.
3.4. What is the role of granularity in product-line implementation? How is it related to the classification of Table 3.1? Does granularity interact with the dimensions of variability implementations or other quality criteria?
3.5. Select a small existing application, for example LlamaChat.[3] Investigate what extensions (potential features) you could add without making any changes to the existing source code. Which extensions could you additionally provide with only

[3] http://sourceforge.net/projects/llamachat/

small changes to existing source code (changing a line or two). What extensions would require more invasive changes?

3.6. Provide examples of preplanning activities.

3.7. How is feature traceability achieved—if at all—by implementation approaches as classified in Table 3.1? Is separation of concerns a prerequisite for feature traceability?

3.8. In the application selected for Exercise 3.5, select a configurable feature from the documentation (for example, message encryption in a chat server or the ability for users to create their own channels). Locate all corresponding source code in the implementation. How did you trace the feature to its implementation? Does the implementation contain any mechanisms that supported you in tracing the feature? What granularity has the extension that implements the feature?

3.9. What is the role of separation of concerns and information hiding in product-line development? Are they necessary? What do we gain from them?

3.10. Is the feature you selected in Exercise 3.8 separated in any form? Can you identify information hiding? What information is hidden and how?

3.11. What is the practical relevance of the principle of uniformity?

3.12. What code and noncode languages are involved in the implementation of the application chosen for Exercise 3.5? Does variability affect multiple languages? Should it? If so, name implementation artifacts in which variability would be necessary.

3.13. Investigate how variability is implemented in the following projects and classify the implementation strategy with regard to the concepts discussed in this chapter:

(a) the Linux kernel (for example, drivers),
(b) the Eclipse development environment (for example, plug-ins for new languages),
(c) the Apache webserver (for example, authentication or error document), and
(d) the application selected for Exercise 3.5.

Discuss why do you think that the developers have chosen the respective techniques. Would there be alternatives?

3.14. Revisit the three scenarios from Exercise 2.9. Discuss for each scenario which quality criteria is the most important and the least important. Explain why you would (or would not) judge importance differently for the three scenarios?

Chapter 4
Classic, Language-Based Variability Mechanisms

After reading the chapter, you should be able to

- implement features with run-time parameters, white-box frameworks, black-box frameworks, and components,
- discuss trade-offs between these implementation techniques,
- select a suitable implementation technique for a given product line,
- choose suitable design patterns to implement variability,
- explain limitations of inheritance and possible solutions, and
- decide what size is appropriate for a component in a product line.

There are many ways to implement variable code; some have been used long before the advent of software product lines. Even a simple if statement offers a choice between different execution paths. To prevent cluttering of code with if statements, to enhance feature traceability, to provide extensibility without the need to change the original source code, and to provide compile-time (or load-time) variability, developers have identified many common programming patterns to support variability.

In this chapter, we discuss four implementation techniques in detail: parameters (Sect. 4.1), design patterns (Sect. 4.2), frameworks (Sect. 4.3), and components and services (Sect. 4.4). All of them can be encoded in mainstream programming languages. In Chap. 5, we discuss approaches based on configuration-management tools (version control systems, build systems, and preprocessors) that operate on top of source code (see Sect. 3.1.2, p. 49) to achieve and manage variability.

The language mechanisms and tools discussed in this and the next chapter are well-known and commonly used in practice. Most industrial software product lines are implemented with one or more of them. As we will see, each has distinguishing properties and gives rise to trade-offs, which we discuss in terms of the classifications (binding times, granularity, and so forth) introduced in Chap. 3.

4.1 Parameters

A simple way to implement variability is to use conditional statements (such as if and switch) to alter the control flow of a program at run time. In our context, conditional statements are typically controlled by configuration parameters passed to a method or a module, or set as global variables in a system. Different parameter assignments lead to different program executions.

There are many ways to set configuration parameters. Often, command-line parameters (such as in "ssh -v") or configuration files (such as system.ini or httpd.conf) are read at startup and stored in global variables. Also, users can change parameters in preference dialogs, sometimes with immediate effects, other times requiring a restart. Furthermore, values of variables can also be hard-wired in source code, so changing them requires recompilation.

Of course, configuration parameters do not have to be stored in global variables. It is quite common to pass configuration parameters as method arguments. Sometimes, configuration parameters are propagated from method to method, or from class to class, across the entire source code. Global variables are convenient to access, avoiding the need for additional method parameters, but they also discourage a modular solution, in which each module or method describes the configuration parameters it expects as part of its interface.

In a feature-oriented setting, we expect one Boolean parameter per feature. In a disciplined implementation, the relationship between features and parameters is expressed and enforced by naming conventions and thus easy to trace (see Sect. 3.2.2, p. 54).

Example 4.1 In Fig. 4.1, we show our graph example with two configurable features, Weighted and Colored, implemented as global configuration parameters. Class Conf stores two parameters (public static is the Java way of specifying a global variable), possibly initialized during startup from command-line parameters or configuration files. These parameters are evaluated inside if statements in the classes Graph, Node, and Edge to trigger feature-specific behavior on demand. □

4.1.1 Discussion

Implementing variability with parameters is straightforward. For this reason, it is widely used in practice. Variation is evaluated at run time when conditional statements are executed (see *load-time and run-time binding* in Sect. 3.1.1, p. 48). The parameter approach shares the usual benefits and drawbacks of run-time binding:

- All functionality is included in all deployed products, even if it is statically known that a feature will never be selected. This has potentially negative implications for resource consumption, performance, and security: First, deactivated functionality is still delivered. Second, performing run-time checks requires additional

```
 1 class Conf {
 2   public static boolean COLORED = true;
 3   public static boolean WEIGHTED = false;
 4 }
 5
 6
 7 class Graph {
 8   Vector nodes = new Vector();
 9   Vector edges = new Vector();
10   Edge add(Node n, Node m) {
11     Edge e = new Edge(n,m);
12     nodes.add(n);
13     nodes.add(m);
14     edges.add(e);
15     if (Conf.WEIGHTED)
16       e.weight = new Weight();
17     return e;
18   }
19   Edge add(Node n, Node m, Weight w) {
20     if (!Conf.WEIGHTED)
21       throw new RuntimeException();
22     Edge e = new Edge(n, m);
23     e.weight = w;
24     nodes.add(n);
25     nodes.add(m);
26     edges.add(e);
27     return e;
28   }
29   void print() {
30     for(int i=0; i<edges.size(); i++){
31       ((Edge) edges.get(i)).print();
32       if(i < edges.size() - 1)
33         System.out.print(" , ");
34     }
35   }
36 }
37 class Node {
38   int id = 0;
39   Color color = new Color();
40   Node (int _id) { id = _id; }
41   void print() {
42     if (Conf.COLORED)
43       Color.setDisplayColor(color);
44     System.out.print(id);
45   }
46 }
47
48
49 class Edge {
50   Node a, b;
51   Color color = new Color();
52   Weight weight;
53   Edge(Node _a, Node _b) {a=_a; b=_b;}
54   void print() {
55     if (Conf.COLORED)
56       Color.setDisplayColor(color);
57     System.out.print(" (");
58     a.print();
59     System.out.print(" , ");
60     b.print();
61     System.out.print(") ");
62     if (Conf.WEIGHTED) weight.print();
63   }
64 }
65
66
67 class Color {
68   static void setDisplayColor(Color c)...
69 }
70 class Weight {
71   void print() { ... }
72 }
```

Fig. 4.1 Graph library: Variability implemented with parameters

computing overhead. In our example of Fig. 4.1, even though we know that we never use feature Colored, the application still contains class Color, evaluates an additional if statement whenever a method print is called, and requires memory for an additional field of every node and edge object. Third, we cannot even prevent others from instantiating objects or invoking methods of deactivated feature code, other than throwing run-time errors (see method add in Lines 19–28 of Fig. 4.1). Finally, shipping unused code opens unnecessary potential targets for attacks, such as buffer-overflow attacks.

- It is possible to alter a feature selection without stopping the program. However, run-time changes are nontrivial in general, as a feature's code may depend on certain initialization steps or assume certain invariants. For example, in Fig. 4.1, we might run into a null-pointer exception, if we enabled Weight at run time, because the field weight of previously created edges was uninitialized. In such cases, it might be easier to require a restart of the program, when configuration parameters change.
- An advantage of passing configuration parameters as method arguments (in contrast to using global variables or using compile-time variability) is that different

parts of the control flow can be configured differently. For example, we could use colored graphs and uncolored graphs in the same program.

It is possible to specialize a program statically when some configuration parameters are known at compile time. Many compilers include optimizations that remove dead code. For example, if we know at compile time that a parameter is always deactivated (for instance, because it is defined as a constant in the source code), the compiler can remove corresponding conditional statements and their bodies. However, deciding when to remove entire methods or classes is less obvious and rarely implemented in contemporary compilers. Compilers also differ in the amount of analysis they perform to recognize dead code when configuration parameters are passed across method boundaries, are modified, or are assigned to other variables. Beyond simple dead-code optimizations, more sophisticated optimizations using partial evaluation can be applied to statically eliminate variability (Jones et al. 1993), but these are far from mainstream or easy to use yet. All in all, despite limited possibilities, the parameter approach is not well-suited for compile-time binding. We will see a specialized form of if statements for compile-time binding later in the context of preprocessors, in Sect. 5.3.

Dependencies between features must be checked when the parameters are configured at startup, or whenever parameters are changed at run time. In our experience, feature dependencies are rarely checked in a systematic way when using parameters. The parameter approach cannot statically guarantee invariants on feature selections (meaning that without considerable effort, it may be possible to activate features at run time that are not compatible with each other).

Adding conditional statements to the source code is a form of annotation (see *annotation versus composition* in Sect. 3.1.3, p. 50). Annotations with conditional statements are available at a fine granularity (see *granularity* in Sect. 3.2.5, p. 59). With if statements, we can change the program behavior at statement level, and many languages even provide conditionals at the expression level (such as, “a?b:c” in Java). Most languages do not provide conditionals at the level of methods or classes, because they are not necessary to influence the behavior of the program (they become relevant only for compile-time variability, which is not the goal of the parameter approach). Extensions are usually performed invasively, but therefore also do not require specific preplanning (see *preplanning* in Sect. 3.2.1, p. 53). Configuration parameters can be encoded in essentially all programming languages; however, they are usually not applicable to noncode artifacts, such as, design documents, grammars, and documentation (see *uniformity* in Sect. 3.2.6, p. 60).

Configuration parameters often lead to implementations that have a poor code quality. On the one hand, global parameters are tempting but reduce modularity (they violate separation of concerns and potentially breach information-hiding interfaces). On the other hand, propagating method arguments requires boilerplate code and may lead to methods with many parameters (considered as code smell by Fowler (1999, pp. 78f)). A typical recommendation is to pass a single configuration object that encapsulates multiple configuration options, known as parameter objects (see, Fowler 1999, pp. 295ff).

Finally, with the parameter approach, variability-related code crosscuts the remaining implementation (see *crosscutting concerns* in Sect. 3.2.3, p. 55). Feature code is scattered across multiple files and modules, in variables, in method arguments, in if statements, and so forth. Furthermore, feature code is tangled with the base code and code of other features. Due to the crosscutting nature, it is difficult to encapsulate a feature's code behind an interface and to place all feature code in one cohesive structure (see *information hiding* in Sect. 3.2.4, p. 57). Due to the scattering and lack of cohesion, it can be nontrivial to trace a feature to all code fragments implementing it (see *feature traceability* in Sect. 3.2.2, p. 54): Unless specific conventions are used, one has to follow the control flow, possible assignments to other variables, and possible operations on the configuration parameters. The parameter approach can lead to undisciplined ad-hoc implementations that are difficult to analyze, maintain, and debug.

Summary parameters
Strong points:

- Easy to use, well-known.
- Flexible and fine grained (see Sect. 3.2.5, p. 59).
- First-class programming-language support (see Sect. 3.2.6, p. 60).
- Different configurations within the same program possible.

Weak points:

- All code is deployed, no compile-time configuration (see Sect. 3.1.1, p. 48).
- Often used in ad-hoc or undisciplined fashion.
- Boilerplate code or nonmodular solutions.
- Scattering and tangling of configuration knowledge (see Sect. 3.2.3, p. 55).
- Separation of feature code and information hiding possible, but left to the discipline of developers (see Sects. 3.2.3 and 3.2.4, p. 55 and 57).
- Extensions typically require invasive changes, but little preplanning though (see Sect. 3.2.1, p. 53).
- No support for noncode artifacts (see Sect. 3.2.6, p. 60).
- Nontrivial tracing between features and code (see Sect. 3.2.2, p. 54), and thus difficult to analyze statically (see Sect. 10.2.3, p. 257).

4.2 Design Patterns

A problem of the parameter approach is that variability is scattered and hard-wired in the source code, often in an undisciplined fashion. Consequently, many patterns have evolved on how variability can be separated and decoupled, among them many well-known *design patterns* for object-oriented programming, such as *observer*, *template method*, *strategy*, *decorator* (Gamma et al. 1995).

Definition 4.1. *Design patterns* are descriptions of collaborating objects and classes that are customized to solve a general design problem in a particular context.

(Gamma et al. 1995) □

Because design patterns are already a part of many curricula, they are increasingly common in practice, and there are many excellent descriptions (most prominently, Gamma et al. 1995), we describe only four design patterns briefly that are well-suited for variability implementations.

4.2.1 Observer Pattern

The *observer pattern* (also known as publish/subscribe pattern) describes a common way to implement distributed event handling, in which a subject notifies all registered observers of changes to its state. The observer pattern decouples subject and observers and adds flexibility to add or remove observers later. For example, a data store (the subject) could notify user-interface elements such as tables, charts, and alerts (the observers) whenever its data changes, independent of how many user-interface elements currently display the data.

The observer pattern consists of three roles: (a) An *observer interface*, which contains one or more methods that are invoked by the subject upon state changes or other events. (b) *Concrete observers*, implementing the observer interface and reacting to changes by the subject. (c) A *subject*, to which observers can register themselves. A subject dispatches events to all registered observers. In Fig. 4.2, we illustrate the architecture and a small schematic implementation.

A subject broadcasts events to all registered observers. Instead of hardwiring the notification mechanism, developers can flexibly add and remove observers at runtime. The subject does not need to know all observers; in fact, the subject only knows the observer interface and has a (dynamically-changing) list of observers.

In product-line development, the observer pattern makes it easy to add and remove features, provided that a feature can be implemented as an observer. Each feature implements the observer interface and registers itself with the subject for relevant events, such as opening a file, sending a mail, committing a transaction, or printing the nodes of a graph. Variability is achieved by registering or not registering observers. Code of different features can be separated into distinct observer classes (see *separation of concerns* in Sect. 3.2.3, p. 55). This way, additional features can be added without changing the implementation of a subject (the part that is common to all products of the product line).

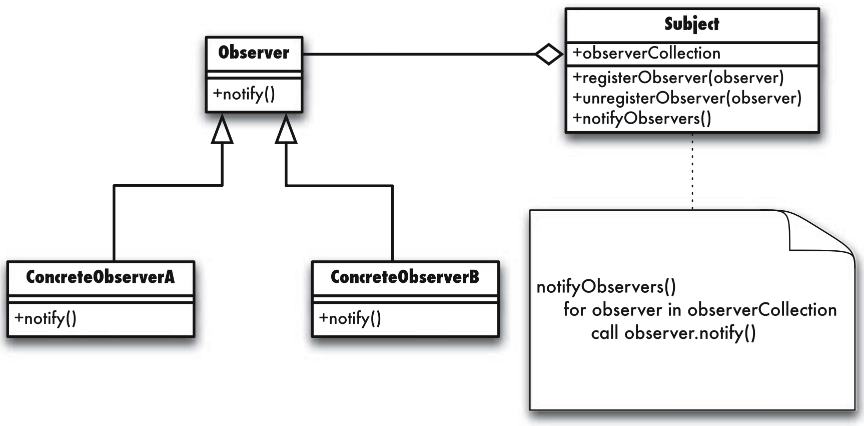


Fig. 4.2 Observer pattern

The observer pattern requires *preplanning* (see Sect. 3.2.1, p. 53). A developer needs to decide upfront where variation will be needed later and to prepare the code, by providing a registration facility and exposing relevant information through the observer interface. Extensions can only be added without invasive modifications of the base code, when the observer pattern was prepared in the base code. Furthermore, additional indirections (for example, calling notifyObservers) cause (a small) architectural run-time overhead, even if no observers are added.

In Fig. 4.3, we adapt the observer pattern to decouple feature code inside the print methods of our graph example. The base implementation of the graph acts as subject, and edges issue notifications when they are printed. These notifications are consumed by the observers WeightPrintObserver and ColorPrintObserver, which implement the parts of the features Weighted and Colored, respectively. Note that the observer mechanism is general; we could use it to implement also other features than printing at the same extension point without changing the base code.

4.2.2 Template-Method Pattern

The *template-method pattern* defines a skeleton of an algorithm in an abstract class, but leaves certain steps of the algorithm to be specified by a subclass. Different subclasses can provide different implementations of these steps by overriding one or multiple methods, and can thus influence program behavior. The pattern exploits subtype polymorphism and late binding in object-oriented programming to execute always the most specific implementation of each method. The template-method pattern is the core mechanism for implementing white-box frameworks, discussed later in Sect. 4.3.1.

```java
class Graph {
  ...
  List<IPrintObserver> observers =
        new ArrayList<IPrintObserver>();
  void register(IPrintObserver o) {
    observers.add(o);
  }
  void unregister(IPrintObserver o) {
    observers.remove(o);
  }
  void notifyBeforePrint(Edge e) {
    for (IPrintObserver o : observers)
      o.beforePrint(e);
  }
  void notifyAfterPrint(Edge e) {
    for (IPrintObserver o : observers)
      o.afterPrint(e);
  }
}

interface IPrintObserver {
  void beforePrint(Edge edge);
  void afterPrint(Edge edge);
}
```

```java
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Graph graph;
  Edge(Node _a, Node _b, Graph _g) {
    a = _a; b = _b; graph = _g;
  }
  void print() {
    graph.notifyBeforePrint(this);
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
    graph.notifyAfterPrint(this);
  }

}

class WeightPrintObserver
      implements IPrintObserver {
  public void beforePrint(Edge edge) { }
  public void afterPrint(Edge edge) {
    edge.weight.print();
  }
}

class ColorPrintObserver
      implements IPrintObserver {
  public void beforePrint(Edge edge) {
    Color.setDisplayColor(edge.color);
  }
  public void afterPrint(Edge edge) { }
}
```

Fig. 4.3 Graph library: Variability in method print implemented following the observer pattern

Implementations of this pattern are straightforward in Java: We implement the algorithm skeleton as one or more methods in an abstract class. For invoking behavior that is subclass-specific, we call corresponding *abstract methods*. Alternatively, we could provide default behavior in virtual but concrete methods that may be overridden by a subclass. Subsequently, a subclass extends the abstract class and provides custom behavior. Different subclasses can provide different specific behaviors, but all share the overall implementation skeleton of the algorithm.

In product-line development, we can exploit this pattern and implement behavior of alternative features by means of different subclasses. Especially, if the algorithm differs only in minor details in each feature, we can share the common parts of the algorithm in a common abstract class. In Fig. 4.4, we illustrate how to implement weighted and unweighted graphs in an excerpt of our graph example. Note that, beyond just overriding existing methods, a subclass can also introduce additional behavior, as with the additional method add in class WeightedGraph.

The template-method pattern is best suited for alternative features (that is, when only one feature out of a set of features can be selected at a time, see Sect. 2.3, p. 26). Also individual optional features can be implemented, when a default implementation is provided for each template method. However, the template-method pattern is not

```java
abstract class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = createEdge(n, m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  protected abstract Edge createEdge(Node n, Node m);
  ...
}

class UnweightedGraph extends Graph {
  protected Edge createEdge(Node n, Node m) {
    return new Edge(n, m);
  }
}

class WeightedGraph extends Graph {
  protected Edge createEdge(Node n, Node m) {
    WeightedEdge e = new WeightedEdge(n, m);
    e.weight=new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    WeightedEdge e = (WeightedEdge) createEdge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
}
```

Fig. 4.4 Graph library: Variability between weighted and unweighted graphs with the template-method pattern

suited for combining multiple features, due to limitations of inheritance (see the discussion in Fig. 4.9, p. 78).

Similar to the other patterns, the template-method pattern separates feature code from base code (see *separation of concerns* in Sect. 3.2.3, p. 55). Feature code is placed in distinct classes and induces a moderate run-time overhead, due to additional invocations of virtual methods. Some authors classify variation through the template-method pattern as a distinct implementation strategy 'inheritance' or 'subtype polymorphism' (Anastasopoules and Gacek 2001; Muthig and Patzke 2002).

4.2.3 Strategy Pattern

The *strategy pattern* aims at variability in algorithms, similar to the template-method pattern. The strategy pattern is different in that it uses delegation instead of inheritance. Instead of writing an abstract method to be overridden by clients, a

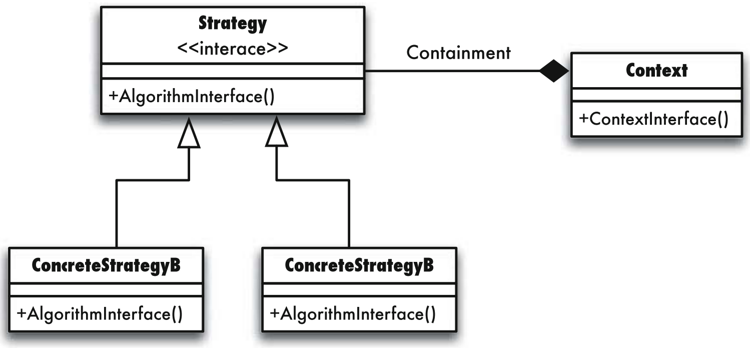


Fig. 4.5 Strategy pattern

developer specifies a strategy interface that is implemented by clients. The strategy pattern encodes a *callback* mechanism and is the core mechanism for implementing black-box frameworks, as discussed in Sect. 4.3.2.

The strategy pattern consists of three roles, as illustrated in Fig. 4.5: The *context* that implements the main algorithm (comparable to the abstract class in the template-method pattern, or the subject in the observer pattern); a *strategy interface* that describes the functionality that can be provided by clients (similar to the observer interface); and *concrete implementations* of the strategy. A strategy is passed to the context in some form, for example, as a constructor parameter, with a setter method, or as method argument; the context may call the strategy's methods.

Using the strategy pattern, a client can select which implementation of the strategy should be used during the execution. In this way, it is easy to add subsequently new implementations.

In product-line development, the strategy pattern is well-suited to implement alternative features, provided that features correspond to different implementations of methods. The pattern replaces ad-hoc conditional statements in the source code with polymorphic calls to the strategy interface. The language dispatches the method call to the concrete strategy implementation. Implementing features as a strategy encourages programmers to encapsulate features with interfaces in a disciplined form (see *information hiding* in Sect. 3.2.4, p. 57). Developers can precisely specify the interface for alternative implementations and future variations. In Fig. 4.6, we show the weighted-versus-unweighted decision from the template-method example, implemented using the strategy pattern.

Although the strategy pattern is best suited for alternative features, developers can also encode optional features. To that end, developers provide default or dummy implementations of strategies for deselected features or accept *null* as strategy parameter. Finally, combining multiple features is possible, if prepared accordingly: we could accept multiple strategies and execute them all (and potentially pass the result of one strategy as input for the next); we show an example later with filter plug-ins in Sect. 4.3.3.).

```
interface IEdgeStrategy {
  Edge createEdge(Node n, Node m);
}

class Graph {
  final private IEdgeStrategy strategy;
  Vector nodes = new Vector();
  Vector edges = new Vector();

  Graph(IEdgeStrategy _strategy) {
    strategy = _strategy;
  }
  Edge add(Node n, Node m) {
    Edge e = strategy.createEdge(n, m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  ...
}

class WeightedEdgeStrategy implements IEdgeStrategy {
  public Edge createEdge(Node n, Node m) {
    WeightedEdge e = new WeightedEdge(n, m);
    e.weight = new Weight();
    return e;
  }
}

class UnweightedEdgeStrategy implements IEdgeStrategy {
  public Edge createEdge(Node n, Node m) {
    return new Edge(n, m);
  }
}
```

Fig. 4.6 Graph library: Weighted and unweighted graph variability with the strategy pattern

The strategy pattern encourages encapsulation and decoupling of features, and even enables distributed development and separate compilation. We discuss benefits and drawbacks of variability with the strategy pattern in more detail in the context of frameworks, in Sect. 4.3.5.

4.2.4 Decorator Pattern

The *decorator pattern* (also known as the wrapper pattern) describes a delegation-based mechanism to flexibly extend objects with additional behavior. Decorators enable objects of a (prepared) class to be extended with additional behavior at run time. Multiple extensions can be combined. The delegation-based decorator pattern can elegantly solve some composition problems that are problematic with inheritance. It can be seen as a dynamic and restricted form of mixin composition (see also Fig. 4.9 and Sect. 6.1.3).

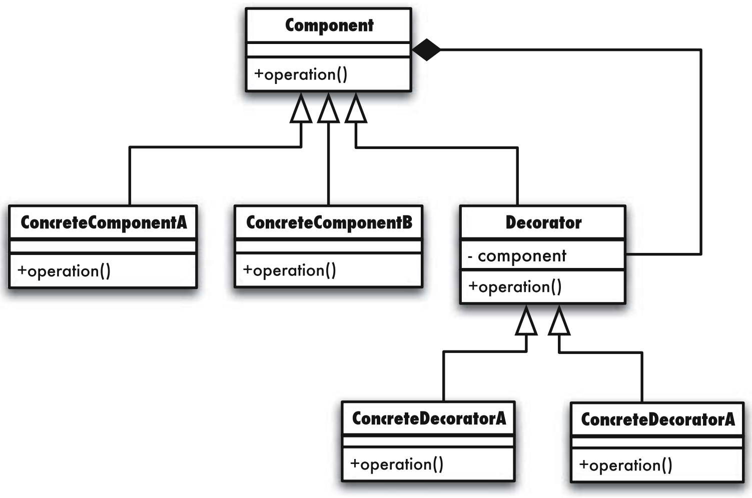


Fig. 4.7 Decorator pattern

According to the terminology of the decorator pattern, the extensible class is called a *component*. The decorator pattern consists of four roles, as illustrated in Fig. 4.7: The *component interface* that describes the (extensible) behavior of the component, a concrete implementation of the *component*, optionally, an abstract *decorator class*, and one or more concrete *decorator implementations*. The concrete component implementations and all decorators implement the component interface. The decorators receive a component as a constructor argument and forward all calls to that component, except for the intended changes to the behavior. For example, a decorator can intercept selected method invocations, whereas it forwards all remaining invocations to the decorated component. Decorators are added to a component object o by wrapping around it (for example, o = **new** DecoratorA(o);).

The strength of decorators is that additional behavior can be added incrementally at run time to existing classes. Furthermore, a series of decorators can wrap the same class, integrating the wrapper and class functionality. To the outside world, the class always provides the same interface, decorated or not. Probably the best known application of the decorator pattern in the Java world is streams in Java's standard library. Input streams all share a common interface, but there are multiple concrete implementations, such as ByteArrayInputStream and FileInputStream. The concrete implementations can be extended with several optional decorators, such as BufferedInputStream, CipherInputStream, and AudioInputStream. A developer can flexibly select core implementation and decorators, even at run-time. For example, in the following code fragment each decorator adds additional functionality to the methods of FileInputStream:

```
InputStream str = new  BufferedInputStream(
                            new CipherInputStream(
                                new FileInputStream(file)));
```

```
class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  IEdge add(Node n, Node m) {
    IEdge e = new Edge(n, m);
    if (Conf.COLORED)
      e=new ColoredEdge(e);
    if (Conf.WEIGHTED)
      e=new WeightedEdge(e);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  ...
}

interface IEdge {
  void print();
}

class Edge implements IEdge {
  Node a, b;
  Edge(Node _a, Node _b){a=_a; b=_b;}
  public void print() {
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
  }
}
abstract class EdgeDecorator
      implements IEdge {
  protected IEdge edge;
  EdgeDecorator(IEdge _edge) {
       edge=_edge; }
  public void print() {
    edge.print();
  }
}

class ColoredEdge extends EdgeDecorator {
  Color color = new Color();
  ColoredEdge(IEdge edge) {
    super(edge);
  }
  public void print() {
    Color.setDisplayColor(color);
    super.print();
  }
}

class WeightedEdge extends EdgeDecorator{
  Weight weight;
  WeightedEdge(IEdge edge) {
    super(edge);
    weight=new Weight();
  }
  public void print() {
    super.print();
    weight.print();
  }
}
```

Fig. 4.8 Graph library: Decorators for the features Weighted and Colored

In product-line development, the decorator pattern is well-suited to implement optional features and feature groups of which multiple features can be selected. In Fig. 4.8, we illustrate decorators by implementing extensions of the features Weighted and Colored to class Edge (similar to the observer-pattern example, in Fig. 4.3). Note that we hard-code the installation of decorators in method addEdge depending on configuration parameters. Of course, we could use also the strategy pattern or other mechanisms to customize which decorators to install. We could even add decorators to existing objects in a running system.

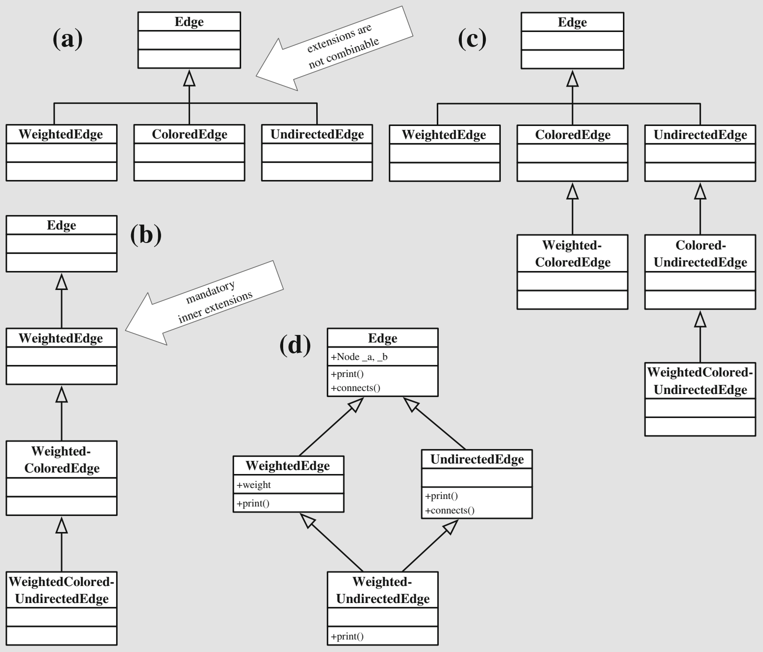
4.2.5 Discussion

Design patterns offer general solutions to reoccurring design problems. Implementing variability is a reoccurring problem, and several patterns provide guidance, as we have shown. Design patterns provide building blocks that are often adopted, used combined, or used in a larger context, for example, as part of frameworks, which we discuss in Sect. 4.3.

The advantages and drawbacks of design patterns are similar to those of the parameter solution, discussed in Sect. 4.1.1; here, we focus on the differences. In general,

Limitations of Inheritance: With inheritance, developers can decouple optional or alternative extensions from the base class. However, in product-line development, inheritance can quickly become limiting when multiple features should be combined.

Consider the following example from the graph library. We want to separate class `Edge` from its extensions `WeightedEdge`, `ColoredEdge`, and `DirectedEdge`. A first possibility (Example a) is to let all three extensions inherit directly from class `Edge`; however, now we cannot combine the extensions, for example, to create edges that are both colored and weighted. Alternatively, we could let all extensions form a linear inheritance hierarchy (Example b). Still, we cannot freely combine extensions; we have to create chains of subclasses for every combination of extensions (Example c), leading to massive code replication (Smaragdakis and Batory, 2002).



Multiple inheritance can offer a possible way out: A class can inherit from multiple superclasses. As illustrated in Example d, we could create a class `WeightedColoredEdge` that inherits both from `WeightedEdge` and `ColoredEdge`. However, if both superclasses have changed the same method, we run into the well-known *diamond problem* (Snyder, 1986): Which of the two possible super methods should be invoked. Due to the diamond problem, multiple inheritance is supported only in few languages.

Another solution is *mixin composition*: a restricted form of multiple inheritance. Mixin composition receives increasing attention with support in languages such as Scala and Ruby, but is not available in mainstream languages such as Java or C++. We will see work-arounds to the limitations of inheritance in Chapter 6.

Fig. 4.9 Limitations of Inheritance with regard to compositional flexibility

design patterns provide good-practice guidelines for disciplined implementations of variability. They are well-known and facilitate communication between developers. In contrast to native ad-hoc implementations with parameters, they encourage decoupling and encapsulation of features (see *separation of concerns* and *information hiding* in Sects. 3.2.3 and 3.2.4, p. 55 and 57) and support a clear tracing of features to observers, subclasses, strategies, decorators and others (see *feature traceability* in Sect. 3.2.2, p. 54). Design patterns enable noninvasive future extensions without changing the base implementation, at the cost of some preplanning effort (see *preplanning effort* in Sect. 3.2.1, p. 53). Since the patterns describe only designs—not concrete code snippets—they can be encoded in different programming languages, but not in noncode languages (*uniformity* in Sect. 3.2.6, p. 60). However, most implementations of design patterns add boilerplate code and architectural overhead, which may influence binary size and performance negatively.

All design patterns discussed here enable variability at run-time (see *binding times* in Sect. 3.1.1, p. 48). For some languages, there are also encodings of these patterns that allow compile-time specialization (such as inlining and static binding) for cases when the configuration choice is already known at compile-time. For example, Czarnecki and Eisenecker (2000, Chap. 7) discuss how to encode design patterns efficiently with generic-programming techniques in C++.

Summary design patterns. Similar to parameters (see Sect. 4.1.1, p. 66), but with the following distinctions.

Strong points:

- Well established, ease communication between developers.
- Guidelines for disciplined design.
- Separate feature code from base code (see Sect. 3.2.3, p. 55), possibly with clear interfaces (see Sect. 3.2.4, p. 57).
- Noninvasive extensions without modifying the base code, given a preplanning effort (see Sect. 3.2.1, p. 53).

Weak points:

- Boilerplate code and architectural overhead.
- Preplanning of extension points necessary (see Sect. 3.2.1, p. 53).

4.3 Frameworks

A *framework* is an incomplete set of collaborating classes that can be extended and tailored for a specific use case. It represents a reusable base implementation for a related set of problems, and thus perfectly fits the needs of product-line development.

A framework provides explicit points for extensions, called *hot spots*, at which developers can extend the framework. Often, extensions are called *plug-ins*. In the same manner as the template-method design pattern (see Sect. 4.2.2, p. 71) and the strategy design pattern (see Sect. 4.2.3, p. 73), a framework is responsible for the main control flow and asks its extensions for custom behavior, on demand; a principle called *inversion of control* (Johnson and Foote 1988).

Nowadays, frameworks with plug-ins are popular in end-user software, including web browsers, graphics-editing software, media players, and *integrated development environments (IDEs)*. For example, the Eclipse IDE is a framework (actually a set of many frameworks) that can be tailored with thousands of plug-ins (Gamma and Beck 2003). In Eclipse and all other frameworks, the basic application is extensible with specific plug-ins. Furthermore, plug-ins are often developed and compiled independently by third-party developers.

In feature-oriented product-line development, ideally, we develop one plug-in per feature and configure the application by assembling and activating plug-ins corresponding to the feature selection—a composition process (see *annotation versus composition* in Sect. 3.1.3, p. 50).

Definition 4.2. A *framework* is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. A framework is open for extension at explicit *hot spots*.

(Johnson and Foote 1988) □

Although historically frameworks predate design patterns, technically, they can be best explained with the design patterns introduced in the previous section. Researchers distinguish between two kinds of frameworks: white-box and black-box (Johnson and Foote 1988). The latter are well-known for using plug-ins.

4.3.1 White-Box Frameworks

White-box frameworks consist of a set of concrete and abstract classes. To customize their behavior, developers extend white-box frameworks by overriding and adding methods through *subclassing*. A white-box framework can be best thought of as a class containing one or more template-methods (see Sect. 4.2.2, p. 71) that developers implement or overwrite in a subclass (actually, it consists of multiple classes).

The "white-box" in white-box framework comes from the fact that developers need to understand the framework's internals. Developers need to identify template methods and can additionally override all other accessible methods. Extensions in white-box frameworks can usually directly access the state of superclasses. All nonprivate fields and methods can be regarded as hot spots of the framework. Extensions in white-box frameworks are usually compiled together with the framework code.

On the one hand, the ability to override existing behavior provides additional flexibility to implement unforeseen extensions. On the other, white-box frameworks require detailed understanding of internals and do not clearly encapsulate extensions from the framework; thus, they are criticized for neglecting modularity.

Typical examples of white-box frameworks are libraries of graphical user interfaces, such as *Swing* or *MFC*, and extensible compilers, such as *abc* or *Polyglot*. We provide a concrete code example, after discussing black-box frameworks.

When using white-box frameworks for product-line variability, we can only add one subclass at a time to a given class, but not mix and match multiple extensions (as explained for the *template-method design pattern* in Sect. 4.2.2). Hence, as the template-method design pattern, white-box frameworks are best suited for implementing alternative features.

4.3.2 Black-Box Frameworks

Black-box frameworks separate framework code and extensions through interfaces. An extension of a black-box framework can be separately compiled and deployed and is typically called a *plug-in*. In feature-oriented product-line development, ideally, each feature is implemented by a separate plug-in.

Definition 4.3. A *plug-in* extends hot spots of a black-box framework with custom behavior. A plug-in can be separately compiled and deployed. □

Whereas white-box frameworks can be understood in terms of the template-method pattern, black-box frameworks follow the strategy and observer patterns (see Sect. 4.2, p. 69). The framework exposes explicit hot spots, at which plug-ins can register observers and strategies. That is, instead of subclassing, black-box frameworks register objects and callback functions. As discussed for the corresponding design patterns, it is possible to provide hot spots that can be extended with multiple plug-ins.

Black-box frameworks are called "black-box" because, ideally, developers need to understand merely their interfaces, but not the internal implementation of the framework. In contrast to a white-box framework, the interface (set of hot spots) of a black-box framework is explicit. Extensions can access only state of the framework that exposed in the interface. Developers can add extensions only to hot spots foreseen (or preplanned) by the framework developer. Although restricting extensions to an interface may limit flexibility, it enables a strict decoupling of framework code and extension code. Furthermore, it can make the framework easier to understand and use, because only a comparably small amount of interface code must be understood.

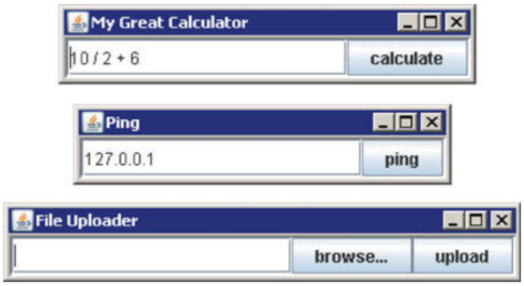


Fig. 4.10 Three similar applications (calculator, ping, and file loader) that can be implemented on *top* of a common framework

The decoupling of extensions encourages separate development and independent deployment of plug-ins, as known from many application-software frameworks, including web-browsers or development environments. As long as the plug-in interfaces remain unchanged, framework and plug-ins can evolve independently.

4.3.3 An Implementation Example for Frameworks

We illustrate the implementation of white-box and black-box frameworks in Java by means of a small example. In Fig. 4.10, we show screenshots of three applications that perform different tasks (calculator, ping, and file loader), but have a similar (trivial) user interface. Their implementations share a relatively large amount of source code for initializing the user interface (fields, buttons, and layout), for starting and stopping the application, and so forth. From the code of the calculator application in Fig. 4.11, only the underlined code fragments differ between the applications, the rest is shared. We demonstrate how to implement the common behavior in a framework and extend it with specific plug-ins, for example, to get the three applications.

A white-box framework is shown in Fig. 4.12: We replace variable code fragments by abstract methods (or overridable methods with default implementations), following the template-method pattern (see Sect. 4.2.2, p. 71). For each extension, we create a subclass and implement the abstract methods to specify custom behavior. The extensions can directly access protected and public methods of the framework, such as getInput in Line 44.

A black-box framework of the same design is listed in Fig. 4.13. Here, we decouple framework and extensions (plug-ins) with an interface Plugin. The extension does not subclass the framework, but implements only the interface. Note the similarity to the strategy pattern (see Sect. 4.2.3, p. 73): The interface Plugin represents a strategy interface called from the context in class App, whereas class CalcPlugin is a concrete strategy.

```java
class Calc extends JFrame {
  private JTextField textfield;
  public static void main(String[] args) { new Calc().setVisible(true); }
  public Calc() { init(); }
  protected void init() {
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
    JButton button = new JButton();
    button.setText("calculate");
    contentPane.add(button, BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText("10 / 2 + 6");
    textfield.setPreferredSize(new Dimension(200, 20));
    contentPane.add(textfield, BorderLayout.WEST);
    button.addActionListener(/* code to calculate */);
    this.setContentPane(contentPane);
    this.pack();
    this.setLocation(100, 100);
    this.setTitle("My Great Calculator");
    // code for closing the window
  }
}
```

Fig. 4.11 Example code of the calculator application. Specifics for the calculator are highlighted

```java
abstract class App extends JFrame {
  protected abstract String
        getApplicationTitle();
  protected abstract String
        getButtonText();
  protected String getInititalText() {
    return "";
  }
  protected void buttonClicked() { }
  private JTextField textfield;
  public App() { init(); }
  protected void init() {
    JPanel contentPane =
        new JPanel(new BorderLayout());
    contentPane.setBorder(new
        BevelBorder(BevelBorder.LOWERED));
    JButton button = new JButton();
    button.setText(getButtonText());
    contentPane.add(button,
         BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText(getInititalText());
    textfield.setPreferredSize(
        new Dimension(200, 20));
    contentPane.add(textfield,
         BorderLayout.WEST);
    button.addActionListener(
        ... buttonClicked(); ...);
    this.setContentPane(contentPane);
    this.pack();
    this.setLocation(100, 100);
    this.setTitle(getApplicationTitle());
    // code for closing the window
  }
  protected String getInput() {
    return textfield.getText();
  }
}
```

```java
class Calculator extends App {
  protected String getButtonText() {
    return "calculate";
  }
  protected String getInititalText() {
    return "(10 - 3) * 6";
  }
  protected void buttonClicked() {
    JOptionPane.showMessageDialog(this,
        "The result of " + getInput() +
        " is " + calculate(getInput()));
  }
  private String calculate(String input){
    ...
  }
  protected String getApplicationTitle(){
    return "My Great Calculator";
  }
  public static void main(String[] args){
    new Calculator().setVisible(true);
  }
}
```

```java
class Ping extends App {
  protected String getButtonText() {
    return "ping";
  }
  protected String getInititalText() {
    return "127.0.0.1";
  }
  ...
  public static void main(String[] args){
    new Ping().setVisible(true);
  }
}
```

Fig. 4.12 White-box framework for single-button applications and two extensions of the framework

```
 1 interface Plugin {
 2   String getAppTitle();
 3   String getButtonText();
 4   String getInititalText();
 5   void buttonClicked() ;
 6   void register(InputProvider app);
 7 }
 8 interface InputProvider {
 9   String getInput();
10 }
```

```
11 class App extends JFrame
12           implements InputProvider {
13   private JTextField textfield;
14   private Plugin plugin;
15   public App(Plugin p) {
16     this.plugin=p;
17     p.register(this);
18     init();
19   }
20   protected void init() {
21     JPanel contentPane =
22         new JPanel(new BorderLayout());
23     contentPane.setBorder(new
24         BevelBorder(BevelBorder.LOWERED));
25     JButton button = new JButton();
26     button.setText(plugin.getButtonText());
27     contentPane.add(button,
          BorderLayout.EAST);
28     textfield = new JTextField("");
29     textfield.setText(
30         plugin.getInititalText());
31     textfield.setPreferredSize(
32         new Dimension(200, 20));
33     contentPane.add(textfield,
          BorderLayout.WEST);
34     button.addActionListener(
35         ... plugin.buttonClicked(); ...);
36     this.setContentPane(contentPane);
37     //...
38   }
39   public String getInput() {
40     return textfield.getText();
41   }
42 }
```

```
43 class CalcPlugin implements Plugin {
44   private InputProvider ip;
45   public void register(InputProvider i) {
46     this.ip = i;
47   }
48   public String getButtonText() { return
         "calculate"; }
49   public String getInititalText() {
         return "10 / 2 + 6"; }
50   public void buttonClicked() {
51     JOptionPane.showMessageDialog(null,
52       "The result of " +
53       ip.getInput() + " is " +
54       calculate(ip.getInput())); }
55   public String getAppTitle() { return
         "My Great Calculator"; }
56   private String calculate(String m) ...
57 }
```

```
58 class CalcStarter {
59   public static void main(String[] args){
60     new App(new CalcPlugin()).
61         setVisible(true);
62   }
63 }
```

Fig. 4.13 Black-box framework for single-button applications and a plug-in

Recall, in a black-box framework, the extension cannot access internals of the framework. To allow extensions accessing information from the framework, we need to provide a callback mechanism. In our example, the framework registers itself to the extension (Line 17), such that the extension can access methods from the framework (Line 53). We even decouple the callback with an additional interface InputProvider to protect what information the framework exposes. Such a callback is not always needed and must not necessarily be implemented with an additional interface as in our example.

To provide a hot spot that can be extended with multiple plug-ins, we store a list of plug-ins instead of a single plug-in (similar to the observer pattern, see Sect. 4.2.1, p. 70). Further, instead of providing a single plug-in interface for all variability, we provide multiple specialized plug-in interfaces. We illustrate both extensions

```
public class App {
  private List<EncoderPlugin> encoders;
  private List<FilterPlugin> filters;
  public App(List<EncoderPlugin> encoders,
       List<FilterPlugin> filters) {
    this.encoders=encoders;
    for (EncoderPlugin plugin: encoders)
      plugin.register(this);
    this.filters=filters;
  }
  public Message processMsg (Message msg) {
    for (EncoderPlugin plugin: encoders)
      if (plugin.canProcess(msg))
        msg = plugin.encode(msg);
    boolean isVeto = false;
    for (FilterPlugin plugin: filters)
      isVeto = isVeto || plugin.veto(msg);
    ...
    return msg;
  }
}
```

Fig. 4.14 Framework that supports multiple plug-ins of different kinds

in Fig. 4.14: Encoders and filters have different plug-in interfaces and the framework accepts a list of both plug-ins; all plug-ins work together to encode and filter messages.

Observe the *inversion of control* that is typical for frameworks. The framework controls the execution and only calls the extension when it requires information.

4.3.4 Loading Plug-Ins

A final question is how to load extensions, especially, plug-ins in black-box frameworks. In white-box frameworks, we simply pass the desired subclass or invoke its main method. In our black-box framework example, we passed the desired plug-in as constructor parameters to the framework from a separate starter class (class CalcStarter in Fig. 4.13). In practice, separate plug-in loaders are common.

In Fig. 4.15, we illustrate a simple plug-in loader for our black-box framework. The loader expects a command-line parameter naming the plug-in class. The loader then uses Java's reflection mechanism to dynamically load the class and instantiate the framework with it.

Beyond this simple example, a plug-in loader typically searches for plug-ins in a certain directory or loads plug-ins listed in a configuration file. Subsequently, the plug-in loader sets up the framework with the corresponding loaded plug-ins. The loader decouples framework and plug-ins even further, as the plug-in loader identifies and loads plug-ins during startup. No further code must be written to activate a plug-in, in the simplest case it is just copied to a specific directory.

Plug-in loaders may additionally check that the plug-in implements required interfaces and check dependencies or ordering constraints between plug-ins. Thus, invalid

```java
public class Starter {
  public static void main(String[] args) {
    if (args.length != 1)
      System.out.println("Plugin name not specified");
    else {
      String pluginName = args[0];
      try {
        Class<?> pluginClass = Class.forName(pluginName);
        Plugin plugin = (Plugin) pluginClass.newInstance();
        new App(plugin).setVisible(true);
      } catch (Exception e) {
        System.out.println("Cannot load plugin " + pluginName + ", reason: " + e);
      }
    }
  }
}
```

Fig. 4.15 Simple plug-in loader using the Java reflection API

plug-in combinations can be rejected at load-time. End-user applications also often provide sophisticated mechanisms to install, update, deactivate, or configure plug-ins, often with graphical front-ends.

4.3.5 Discussion

Frameworks, especially, black-box frameworks, are a suitable way to implement variability in product lines. Using typical plug-in loaders, individual products are created by *composing* plug-ins at *load-time*, but in principle run-time changes are possible (see *binding times* and *annotation versus composition* in Sects. 3.1.1 and 3.1.3, p. 48 and 50). Much like design patterns, frameworks are general in that they can be implemented in most programming languages, but not in noncode languages (XML, documentation, and so forth; see *uniformity* in Sect. 3.2.6, p. 60).

When features are implemented as plug-ins, we can trace them directly (see *traceability* in Sect. 3.2.2, p. 54). With plug-ins in black-box frameworks, we can encode alternative as well as optional features in a disciplined way. We can even combine multiple optional features, as illustrated in Fig. 4.14. Developing one plug-in per feature allows us to flexibly select other features and load the corresponding plug-ins. The entire process from a feature selection to a tailored program can be automated. Plug-ins for deselected features do not need to be deployed, so we can potentially reduce binary size. Also white-box frameworks can be used to implement alternative features or a single optional feature, but combining multiple optional features is problematic due to the limitations of subclassing, as discussed in Fig. 4.9 (p. 78).

In contrast to the parameter approach (Sect. 4.1) and an ad-hoc use of design patterns (Sect. 4.2), black-box frameworks facilitate modularity by following well-defined conventions (see *separation of concerns* in Sect. 3.2.3, p. 55). Plug-ins are encapsulated from the framework implementation through clear interfaces. Ideally, an interface is designed not only for a specific extension in mind, but for a whole set of potential extensions. Framework and plug-ins can be changed independently as long as they still adhere to the common interface. In the best case, all code related to a feature is encapsulated in a single plug-in, and it is possible to understand and maintain the feature by looking only at this plug-in's code (see *information hiding* in Sect. 3.2.4, p. 55).

Modularity allows developers to provide third-party plug-ins that can be compiled and deployed independently. This is especially important for *software ecosystem* (Bosch 2009), in which a community of specialized companies or independent developers provides additional features. Such a development model works both for open-source projects and closed-source projects. Well-known examples of are web browsers and development environments, such as *Eclipse* and *Visual Studio*, both consisting of multiple (mostly black-box) frameworks. For example, users of Eclipse can select from many independently developed open source and commercial plug-ins.

However, frameworks are not without difficulties. Creating and maintaining frameworks is a challenging task. The framework designer must anticipate (or preplan) where hot spots are needed and design corresponding template methods or plug-in interfaces. They must *design for change*, which requires an upfront investment (see *preplanning effort* in Sect. 3.2.1, p. 53). If a framework designer chooses not to expose information that extensions need, these extensions are difficult or impossible to build (without invasive refactoring the framework). Designing a framework is often handed to senior developers, because it requires substantial experience and a deep understanding of the domain.

Once hot spots and interfaces have been fixed, they are hard to evolve: Although developers can add new hot spots to the framework, it is not possible to change the plug-in *interface* without invasively changing all existing plug-ins (some of which might be provided by third parties and not available with source code or not even known). The inflexibility to change a framework may slow down future evolution of the product line. Hence, frameworks are better suited for proactive adoption of product lines than for reactive or extractive adoption (see Sect. 2.4, p. 39).

Plug-ins may be reused in many different instantiations of a framework, but, in contrast to components (which we discuss next), they are not intended to be reused across different frameworks. It is highly unlikely that plug-ins for one framework (or product line) can be plug-ins for other frameworks (or product line). The reason is simple: every framework encodes structures, architectural conventions, and implementation details that are specific to it, and that are unlikely to be shared verbatim by any other framework.

Furthermore, frameworks induce both development and run-time overhead. Developers need to write additional code to decouple extensions from the framework, such as interface Plugin in Fig. 4.13. Even if a hot spot is not extended, additional code for the extension point is required. Hence, frameworks often require more source code, result in a larger binary size, and perform slower due to additional indirections. Often the overhead is acceptable, but not always: In high-performance and embedded scenarios, unnecessary overhead can not be tolerated. Furthermore, when a framework exhibits too many hot spots for potential extensions that are never used (a problem named *speculative generality* by Fowler (1999); a common overreaction to experiencing a too narrow framework), artificial overhead can become problematic.

Fine-grained and crosscutting features (see *granularity* and *crosscutting* in Sects. 3.2.5 and 3.2.3, p. 59 and 55) are hard to implement with frameworks. Fine-grained extensions require hot spots for minimal extensions and crosscutting features require many hot spots; both lead to disproportionally complex designs. Whereas, in our calculator example, an extension modified only four code locations, features such as the transaction subsystem in a database system affect many parts of the framework. For fine-grained and crosscutting features, the framework must expose many details of the framework. This is hard to do: interfaces become bloated making new plug-ins harder to understand and build. Then there is the problem of speculative generality mentioned earlier. When there are simply too many hot spots, the benefits of modularity diminish and other variability techniques must be considered.

Overall, frameworks are better suited for coarse extensions that extend few well-defined points in the control flow (see *granularity* in Sect. 3.2.5, p. 59). There is no technical limitation, but the overhead for implementing fine-grained and crosscutting features can become overwhelming. Features such as transaction management in a database system (crosscutting the entire implementation and changing the behavior in many locations in nontrivial ways) are rarely separated into plug-ins.

Summary frameworks. Like the parameter approach, frameworks are language-based (see Sect. 3.1.2, p. 49). However, frameworks differ from parameters in that they are primarily composition-based, not annotation-based (see Sect. 3.1.3, p. 50), and in that variability is usually decided at load-time, not run-time (see Sect. 3.1.1, p. 48).
Strong points:

- Well-suited for implementing variability.
- Automated product derivation by plug-in loading.
- Static tailoring, deploying only selected features (see Sect. 3.1.1, p. 48).
- Modularity by separating features, hiding feature internals, and enabling feature traceability (especially in black-box frameworks; see Sects. 3.2.2–3.2.4, p. 54–57).
- Suitable for open-world development (black-box frameworks only).
- Disciplined implementation, well-known.

Weak points:

- High upfront design effort (see Sect. 3.2.1, p. 53).
- Difficult evolution.
- Potential development, run-time, and size overhead.
- No reuse outside the framework.
- Unsuited for fine-grained and crosscutting features (see Sects. 3.2.5 and 3.2.3, p. 59 and 55).
- No support for noncode artifacts (see Sect. 3.2.6, p. 59).

4.4 Components and Services

As a last classic language-based implementation approach, we discuss components and services (including the notion of web service). Although component-based implementations are common in product-line practice, they lack the automation potential of feature orientation that we aim at. However, as we will see, components can be integrated to some degree with other implementation approaches. Hence, we introduce components briefly and discuss their benefits and limitations.

Definition 4.4. A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

(Szyperski 1997) □

The key idea of a component is to form a modular, reusable unit. A component provides its functionality through an interface, whereas its internal implementation is encapsulated (also for components, there is a white-box versus black-box discussion (Szyperski 1997, Chap. 4); here, we assume black-box components). To reuse components, they are *composed* with other components in different combinations (see *annotation versus composition* in Sect. 3.1.3, p. 50). A class can be seen as a small component that can be reused in many applications; a library of graph algorithms is an example of a larger component, consisting of many classes.

Already more than three decades ago, Parnas (1979) proposed to implement product lines (back then called program families) by encapsulating changing parts and hiding their internals (see *information hiding* in Sect. 3.2.4, p. 57), so that those parts could be exchanged and removed easily. Domain analysis is essential to *design for change* and to identify and separate those parts that differ between products of a product line.

Proponents motivate the use of components for another reason: *building markets*. The idea is that developers can implement and deploy components independently, and compose components from different sources. As a consequence, developers can decide whether to implement their own components or whether to buy and reuse third-party components. Component markets open a new perspective: Developers can focus on their expertise and develop, perfect, and sell an individual component. Others can buy the component and use it in their software, instead of reimplementing the functionality or buying an entire software product that contains the desired functionality, but that is otherwise not tailored to their needs. Components facilitate a best-of-breed approach, in which developers can decide, for each subsystem, whether to buy the best (or cheapest) component from the market or to implement the functionality individually (Szyperski 1997).

```
package modules.colorModule;

//public interface
public class ColorModule {
  public Color createColor(int r, int g, int b) { /* ... */ }
  public void printColor(Color color) { /* ... */ }

  public void mapColor(Object elem, Color col) { /* ... */ }
  public Color getColor(Object elem) { /* ... */ }

  //just one module instance
  public static ColorModule getInstance() { return module; }
  private static ColorModule module = new ColorModule();
  private ColorModule() { super(); }
}
public interface Color { /* ... */ }

//hidden implementation
class ColorImpl implements Color { /* ... */ }
class ColorPrinter { /* ... */ }
class ColorMapping { /* ... */ }
```

Fig. 4.16 Simple example of component managing colors

Of course, the fallacy here is that components are plug-compatible only if they and their interfaces are designed to existing standards. In fact, reusing components from markets is often problematic, because their architectural assumptions mismatch (Garlan et al. 1995). Unless planned together, components are likely incompatible and require significant engineering effort to compose. Domain engineering improves this picture, as we will discuss shortly.

Services, as discussed in the context of service-oriented architecture (Erl 2005), are a special form of software components. A service encapsulates functionality behind an interface, just like a component. Similarly, proponents envision a market of services. Services typically emphasize standardization, interoperability, and distribution. Especially in the popular form of web services, a service is reachable over a standardized Internet protocol and may run on remote servers; that is, it is not necessary to install and integrate a service locally. Even the lookup of a service can occur at run-time, through the use of service registries. In addition, communication between services is standardized, so services written in different languages can exchange messages. To connect services, called *service orchestration*, several specialized tools and (graphical) languages exist, which simplify the composition process. For our discussion of product-line development, we make no further distinction between components and services.

Example 4.2 In Fig. 4.16, we exemplify a small component from our graph library. Assume that storing and printing colors is nontrivial and might be reused in another project outside the product line. We could extract color management into a reusable component (or package). The component's interface exposes a class ColorModule with several public methods and a Java interface Color, whereas other implementation details are hidden. We use Java's scoping and package mechanism to enforce encapsulation. That is, to ensure that other components may not access private implementation details, we use package visibility. Classes ColorImpl, ColorPrinter, and ColorMapping are not public and visible only inside the package, so other components in other packages can only interact with public classes and methods.

A component is independent of a specific application or product line. Developers can reuse it when implementing the color feature for the graph library, but also for other applications. Note that developers need to write custom code to connect the implementation with the component, for example, extra code to call the component's methods. To reuse the component, only the public interface is of interest; implementation details are not accessible. In Java, we can deploy the component separately as class or jar files.[1] □

4.4.1 Sizing Components

Deciding when to build a reusable component and what to include in that component is a difficult design decision. There is a well-known trade-off between reuse and use (Biggerstaff 1994; Szyperski 1997): A large component that provides plenty functionality is easy to integrate and use, but there might be only few applications in which the component fits. In contrast, one might be tempted to build small reusable components that can be combined flexibly (in an extreme case, put every method into a distinct component), but, then, the overhead of using the component becomes discouraging. The smaller the component, the more programming is left to the user that has to connect the components with the base code and with each other; in extreme cases, components are so small that little remains to reuse and to hide behind their interfaces. Szyperski (1997, Chap. 4) rephrases this as the maxim "maximizing reuse minimizes use."

Example 4.3 Consider the common scenario that a developer searches the web for a piece of software. Suppose you find two choices: One that is small (less than 1000 lines of code) with only a fraction of the functionality that you want; and one that is huge (100 000 lines of code) with many extra functionalities that are highly intertwined with the desired functionality. Which one should be selected? Just on intimidation alone, many developers would select the small application as there is a higher potential to understand what it does, while the rest could still be added manually. In contrast, the big one likely makes incompatible architectural assumptions. It would require considerable integration effort and pose considerable risks. That is, most developers would likely reuse less and implement more themselves. □

[1] In our example, we use the *facade* design pattern to provide a concise interface and the *singleton* design pattern to ensure that only one instance of the component exists at at time (Gamma et al. (1995) discuss these patterns in more detail). A developer would invoke a method somewhat like this: ColorModule.getInstance().createColor(...).

In the tension between reuse and use, developers need to strive for a balance of a component that is large enough to provide useful functionality but small enough to be reused in many contexts. Unfortunately, without knowing when and how a component will be reused, even experienced developers may only guess suitable component size. Unsuitable component size has often been claimed a reason for the limited success of market places for components and services.

In product-line development, *domain analysis* solves this dilemma and guides us in how to size a component to balance reuse and use. As described in Sect. 2.2, during domain analysis, a domain expert investigates potential products within the domain and decides which products are in the scope of the product line. With this information, we can decide upfront which functionality will be reused *within* the product line; thus, we can size components accordingly and coordinate their architectural assumptions. That is, in product-line development, developers solve the sizing problem by *preplanning reuse systematically*. For example, when we know that certain functionality is always used together, we can combine it in the same component and make it easier to use; in contrast, when we know that functionality is only used in few products, it should be separated into its own component. In short, *domain analysis helps us to decide how to divide code into components*.

4.4.2 Composing Components

Developing product lines by constructing and composing reusable components was a common strategy, especially, in the early era of product-line engineering (Bass et al. 1998; Krueger 2006). With domain analysis, developers decided which functionality should be reused across multiple products of the product line and designed components accordingly. In application engineering, developers then composed the corresponding components. In principle, one could create one component per feature or map features to components in some other form.

In contrast to plug-ins in frameworks, components are generally *not* designed to be composed automatically. Component-based software engineering pursues a different mind set, in which building units of functionality is the goal; being plug-compatible with other components is typically not a priority. A common goal of the design-for-change strategy is to (potentially) exchange a component by another one with the same interface, but again automation is not the focus.

Given this, to *derive a product* for a given feature selection during application engineering (see Sect. 2.2, p. 19), a developer selects suitable components and then manually writes *glue code* to connect components for every product individually. Module interconnection languages (DeRemer and Kron 1976) and service orchestration (Erl 2005) aim at providing high-level scripting approaches to gluing together components. Although definitely a manual process, implementing product lines with components still offers a huge benefit over constructing each product from scratch. When considering the economical motivation of product lines in Fig. 1.4 (p. 9) from the first chapter, manual effort during application engineering increases costs per product (and hence the slope), but does not change the overall expected benefit of product lines.

Building product lines with components is suitable if feature selection is performed by developers (not customers) and the number of products is low. For example, Phillips builds software for consumer electronics from reusable components (van Ommering 2002). In this case, product-line developers can construct new products from reusable parts. If required, they can also perform customizations beyond what was preplanned as product-line feature (and without propagating the customization back to the feature model). Especially, when publishing only a few distinct preconfigured products or when implementing tailor-made solutions for few customers, the manual effort in application engineering may be negligible or acceptable.

The goal of feature-oriented product-line development is to entirely *automate* product derivation after selecting features in application engineering (see *push-button approach* in Sect. 2.2.4, p. 26). In a component-based approach, there is no generator that would automatically build a product for a given feature selection; manual developer intervention is required. We argue that automated product derivation is essential for effective, large-scale product-line development. Krueger (2006) even claims "application engineering considered harmful" to emphasize the importance of automation.

4.4.3 Components Versus Plug-Ins

Components and plug-ins share many similarities. They both pursue a modular implementation (ideally a module per feature) and hide implementation details behind an interface (see *traceability* and *information hiding* in Sects. 3.2.2 and 3.2.4, p. 54 and 57) and require similar preplanning effort (see Sect. 3.2.1, p. 53).

Their main difference lies in the automation potential and in reuse beyond a product line: A plug-in is always tailored for a specific framework, and, as such, has very specific requirements on its context. In contrast, components can be intended to be reused even outside a product line. At the same time, the tight integration of plug-ins into a framework allows loading plug-ins automatically without additional per-product development. Deriving a product for a feature selection in a framework requires only assembling the corresponding plug-ins, not writing additional glue code, as necessary for components.

Of course, components can be used within frameworks. We can write glue code that adapts a general-purpose component to the plug-in interface of a framework (see also the *adapter* and *bridge* design patterns as described by Gamma et al. (1995)). This way, we can reuse components within a framework and automate their integration based on a feature selection, while we can still reuse the component in different contexts outside the framework.

Components can be encoded in many languages. Ideally, the language should provide some encapsulation mechanism that can hide internals of the component behind an interface, and that enforces the interface mechanically. However, also weaker notions of encapsulation are possible that translate also to noncode artifacts. For example, we can simply separate grammars, design documents, or models into separate artifacts and let developers combine them manually during product derivation.

Apart from automation and uniformity, as components and plug-ins use similar modularity mechanisms, they share similar benefits and limitations. They modularize features and allow compile-time product derivation, deploying only selected functionality (see *binding times* in Sect. 3.1.1, p. 48). At the same time, both components interfaces and plug-in interfaces are difficult to evolve once they are fixed and other (potentially third-party) implementations rely on them. Both may add overhead due to additional indirections and boilerplate code. Most importantly, components have the same limitations regarding fine-grained and crosscutting extensions (see *granularity* and *crosscutting* in Sects. 3.2.5 and 3.2.3, p. 59 and 55). For example, integrating a crosscutting transaction subsystem provided as component into a database will require much glue code.

Summary components and services
Strong points:

- Well-known and established implementation technique.
- Static tailoring, deploying selected features only (see Sect. 3.1.1, p. 48).
- Separation of concerns, information hiding, and feature traceability (see Sects. 3.2.2–3.2.4, p. 54–57).
- Reuse within and beyond the product line.
- Reuse of third-party implementations.
- Reuse in distributed environments, even with features maintained and run by third parties (especially services).
- Uniformly applicable to many languages (see Sect. 3.2.6, p. 60).

Weak points:

- No automated product derivation, glue code is necessary
- Difficult evolution.
- Potential development, run-time, and size overhead.
- Preplanning necessary to size components (see Sect. 3.2.1, p. 53).
- Unsuited for fine-grained and crosscutting features (see *separation of concerns* in Sect. 3.2.5, p. 59).

4.5 Further Reading

The classical language-based implementation approaches discussed in this chapter are frequently used to implement product lines, but rarely discussed explicitly as such in literature.

Especially for the parameter approach, there is little literature. Modularity and the related concepts of encapsulation and coupling are discussed generally by Meyer (1997, Chaps. 3 and 4). Regarding the problem of methods with too many parameters, Fowler (1999) discusses a corresponding code smell and a solution with a refactoring toward parameter objects. More recently, Reisner et al. (2010) and Rabkin and Katz (2011) began exploring the use of configuration options in programs (not necessarily product lines). Reisner et al. (2010) found that in three analyzed programs many configuration options are orthogonal and rarely interact. Rabkin and Katz (2011) found that configuration options are often not consistently documented, which we interpret as an encouragement to use product-line technology to plan variability in a more systematic way.

The book of Gamma et al. (1995) is still the best reference on design patterns and explains patterns in much detail. Some product-line literature (for example, Muthig and Patzke 2002; Anastaspoules and Gacek 2001) distinguishes variability implementations further into techniques based on delegation, inheritance, parametric polymorphism, and so forth. Many of them and their best practices can be explained in terms of design patterns as well. Czarnecki and Eisenecker (2000, Chap. 7) discuss low-overhead design-pattern implementations for static configuration with C++.

Frameworks have a long tradition. The seminal paper "Designing Reusable Classes" (Johnson and Foote 1988) provides an excellent introduction and discusses trade-offs between white-box and black-box frameworks. An impressive example of framework design in practice is the Eclipse development environment. Several books (and web articles) describe Eclipse's architecture and how they make use of design patterns; even though not the newest book on Eclipse, we recommend the introduction by Gamma and Beck (2003).

Finally, Szyperski (1997) provides a detailed introduction into the concept of components, including a discussion of building markets, technology choices, and how to size components. Erl (2005) provides a broad introduction into the philosophy behind service-oriented architectures. High-level languages for composing components can be traced back to the idea of programming-in-the-large by DeRemer and Kron (1976) and have been evolved since with many module interconnection languages, and more recently service-orchestration approaches (Erl 2005). Czarnecki and Eisenecker (2000) discuss in detail how domain engineering helps to preplan and size components for reuse. If not planned accordingly, reuse can be very hard though: Garlan et al. (1995) discuss architectural mismatch, the reason why composing components that have not been designed together is so difficult.

Exercises

4.1. Implement a basic chat system consisting of a server and multiple clients (in the domain discussed in Exercise 2.4, page 43). The clients show messages received from the server and allow users to post messages to the server. The server broadcasts all received messages to all connected clients.

Subsequently, extend this implementation with the following features:

(a) *Colors:* Messages may have a text and background color. The color can be specified in the client when writing a message.
(b) *Authentication:* To connect to the server, the client must provide a username and password.
(c) *Encryption:* Messages are encrypted. Provide at least two optional encryption mechanisms.
(d) *History:* Server and clients keep a log of all received messages. The client can show the last 10 entries of the log in a dialog.

Hint: Reuse existing source code where possible.

4.2. Implement all features of the chat system (Exercise 4.1) using the *parameter* approach, such that all features can be configured at load-time with command-line parameters, with a configuration file, or even with a graphical dialog in the chat client's user front-end. Critically discuss code quality and implementation effort of the resulting system.

4.3. Discuss the potential of design patterns for implementing the chat system (Exercise 4.1 and 4.2). Change the implementation where appropriate. Discuss the influence of this change on the quality criteria introduced in Chap. 3.

4.4. Find an open-source project that can be configured using configuration files or command-line parameters (for example, command-line utilities, web servers, database engines). Study the end-user documentation to find three (Boolean) configuration options that could be considered as features and investigate how those are implemented.

(a) What is the binding time of the configuration option?
(b) Is the configuration option implemented cohesively or is it scattered throughout the source code?
(c) Are global variables used or are parameters propagated? Are design patterns used in the implementation? Discuss traceability, separation of concerns, and information hiding with regard to the implementation of the configuration option.
(d) Discuss whether design patterns could be used to improve the implementation and prepare it for future extensions.

4.5. Implement the chat system (Exercise 4.1) as a framework that can be extended with plug-ins. Provide a plug-in for each feature. Ensure that framework and features can be compiled separately. Provide a simple plug-in loader (see Sect. 4.3.4) so that the configuration can be changed without any modifications to source code. Critically reflect on code quality and implementation effort of the resulting system; consider also the quality criteria from Chap. 3.

4.6. Extend the implementation of Exercise 4.5 with an additional feature *Spam Filter* that rejects all messages that contain words from a blacklist. Review your implementation: Are new extension points necessary? Is it necessary to change the framework or other plug-ins? Can the new plug-in be understood in isolation? Would the same hold for feature *Command-Line Interface* (instead of a graphical user interface) or a feature *File Transfer*?

4.7. Find a software product that is extensible with plug-ins (for example Miranda-IM,[2] Netbeans,[3] or Mozilla Firefox[4]). Study the developer documentation or source code to find possible extension points.

(a) Is the framework implemented as black-box framework, or white-box framework, or as some combination of those? Are design patterns used for extensibility?
(b) Can plug-ins be compiled separately? What mechanism is used to load plug-ins? What is the binding time?
(c) Name three features that can be added noninvasively as plug-ins using existing extension points.
(d) Name three features that cannot be added with existing extension points but would require invasive changes to the framework.

4.8. Decompose the chat application from Exercise 4.1 into reusable components. Build three different chat products out of these components.

4.9. Discuss possible components that could be reused within and beyond a product line of (a) graph algorithms, (b) chat applications, and (c) the scenarios from Exercise 2.5 (page 43). Discuss suitable size of the components and the potential costs of using them.

4.10. Reconsider the scenarios of Exercise 2.9 (page 44). Which implementation approach would you recommend to the developers and why?

4.11. Compare all discussed implementation approaches in terms of (a) modularity, (b) suitability of distributed development with multiple developers and multiple companies, (c) possibility of buying and integrating parts or features developed by third parties, (c) overhead on run-time performance, (d) overhead on binary size, (e) development effort and required skill, and (f) maintainability.

Use your implementations of the chat example (Exercises 4.2, 4.3, 4.5, and 4.8) to support your analysis.

[2] `http://www.miranda-im.org/`

[3] `http://netbeans.org/`

[4] `http://www.mozilla.org/`

Chapter 5
Classic, Tool-Driven Variability Mechanisms

After reading the chapter, you should be able to

- implement compile-time variable software with version-control systems, build systems, and preprocessors,
- discuss trade-offs between language-based and tool-based implementation techniques,
- select a suitable implementation technique (or combination) for a given domain, and
- distinguish lexical from syntactic preprocessors and disciplined from undisciplined annotations.

Besides language-based techniques, which encode variability with available concepts within programming languages (discussed in the previous chapter), external tools can also be used to implement and manage variability. In particular, we discuss version-control systems (Sect. 5.1), build systems (Sect. 5.2), and preprocessors (Sect. 5.3). Together, these tools are also often called *configuration-management tools*.

Whereas language-based approaches focus mostly on run-time variability, tool-driven approaches typically target compile-time variability (see *binding times* in Sect. 3.1.1, p. 48). The goal of compile-time variability is to compile and include only code that is needed; typically, a tool decides which code fragments the compiler receives. We investigate how different tools handle optional and alternative code.

5.1 Version-Control Systems

Version-control systems track changes in source code and other development artifacts to facilitate collaborative development. Popular examples are *CVS*, *Subversion*, *Perforce*, *Visual SourceSafe*, *git*, and *Mercurial*. Version control is a subfield of software configuration management. As a general discipline, it covers management activities around software building and evolution as a whole, including management decisions such as when and how to process change requests and publish releases (Sommerville 2010, Chap. 25). Here, we focus on using version-control systems for product line development.

5.1.1 Terminology

Version-control systems are best known for tracking revisions of source code, that is, *variation over time*. Each revision gets an identifier and possibly a comment explaining the changes. Developers can go back in time to retrieve earlier revisions of the source code and investigate changes (asking questions regarding *when*, *by whom*, and *why*). Revisions are ordered, newer revisions supersede previous ones.

Most version-control systems allow some form of *branching*. In different branches, the same files can be changed independently; changes in one branch do not affect the files in other branches. In this case, we speak of *variants*[1] of the same file. Instead of revisions over time, variants describe *intentional versioning* by developers, sometimes also called *variation in space*. Variants are not ordered and do not supersede each other; they exist in parallel. Each variant can have an independent revision history.

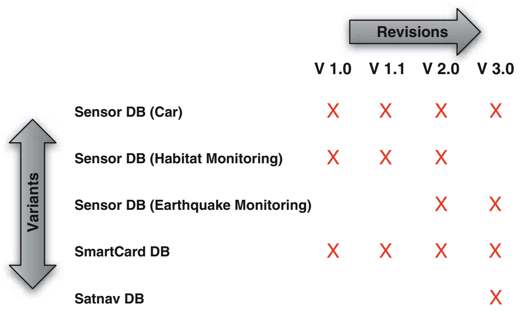
Whereas branching creates a new variant, *merging* combines two variants. Typically, the changes in the revision of one branch are applied to another branch. Merging is typically semiautomated. In practice, merges are mostly performed in a line-based fashion using textual patches, and only when there is a conflict (that is when different changes are applied to the same line) manual intervention is necessary (Mens 2002).

The term *version* denotes both *revisions* and *variants*. Most version-control systems provide a history of revisions and branching of variants.

Closely related to revisions are releases. A *release* is a selected revision of the software that is given a specific name or number and that is deployed to customers. When to release revisions and what release names or numbers to chose, is a management decision. Releases may refer to specific variants or to all variants. Similarly, many version-control systems provide *tags* to label specific versions.

In Fig. 5.1, we illustrate the difference between revisions and variants. Of the same database system (for the scenario see Sect. 1.4, p. 9), there are several revisions (and releases) over time. In our example, version 1.0 is revised to versions 1.1, 2.0, and 3.0. At the same time, there are different variants of the database, one for cars, one for smart cards, one for sensor networks, and so forth. Note that variants may not exist for all revisions and releases. In our example, some variants have been created only in later revisions, whereas some variants are no longer maintained in recent revisions.

[1] As many terms in software engineering, the term *variant* is overloaded with different meanings. Some authors refer to alternative features in feature models and implementations as variants (for example, Pohl et al. 2005), others call the different products of a product line variants (Heidenreich et al. 2010). In this section, we use ‘variants’ as a technical term to distinguish different kinds of versions.



Revisions
V 1.0
V 1.1
V 2.0
V 3.0
Variants
Sensor DB (Car)
Sensor DB (Habitat Monitoring)
Sensor DB (Earthquake Monitoring)
SmartCard DB
Satnav DB

	V 1.0	V 1.1	V 2.0	V 3.0
Sensor DB (Car)	X	X	X	X
Sensor DB (Habitat Monitoring)	X	X	X	
Sensor DB (Earthquake Monitoring)			X	X
SmartCard DB	X	X	X	X
Satnav DB				X

Fig. 5.1 Revisions and variants in a product line of embedded data management systems

We summarize the terminology with the following definition:

Definition 5.1 A *revision* describes ordered variations over time; a *release* is a specifically named revision. A *variant* describes intentional variations that exist in parallel. *Versions* encompass both revisions and variants. □

5.1.2 Building Product Lines with Version-Control Systems

It is appealing to implement a product line by means of multiple branches in a version-control system. Before illustrating how to achieve that, let us look at a typical development process with branching. Many different ways of using branches are established in practice; in Fig. 5.2 we illustrate a common one:

- **Development branch**. The central branch of the project is the development branch in which most development happens.
- **Release branches**. For each planned release, a separate branch is created (branches Release 1 and Release 2 in our example). The separate release branch allows developers to focus on getting a release stable (for example, from beta release to a main release) without interference from the daily ongoing development of the main development branch.
- **Feature branches**. Larger blocks of functionality can also be developed in a separate branch; such feature branches are typically used to separate longer or risky projects, or in our example to develop a feature for a future release. When the feature implementation is finished, it can be merged as a whole into another branch.
- **Bug fixes**. Similar to feature branches, bug fixes can be developed in separate branches and can be merged into one or multiple other branches. In our example, the second bug fix is merged into both release branches in our case and subsequently also into the development branch.

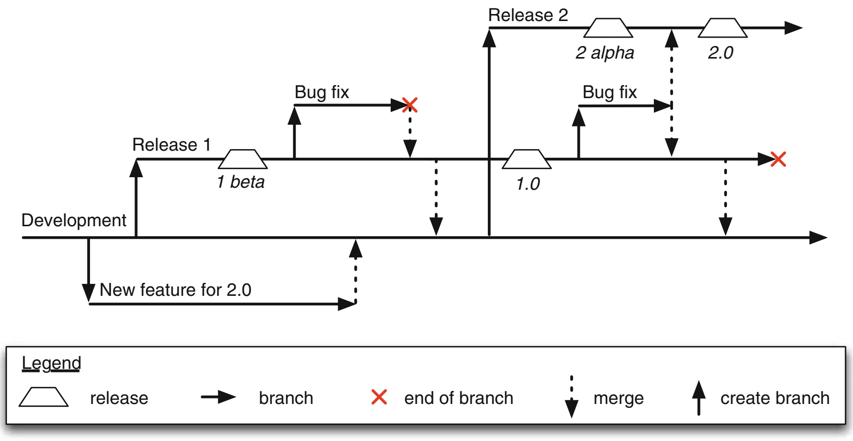


Fig. 5.2 Branching and merging during development and releases

Each of these branches may be terminated at some point, when it is no longer needed. For a discussion of this and other processes, see the broad literature on version-control systems (for example, Chacon 2009 Chaps. 3 and 5).

Developers familiar with branching, often use it to build customer-specific variations, as illustrated in Fig. 5.3. Here, all development of code that is shared by all products happens in a main branch. But upon a release, a new branch is created per customer that requests some modification or extension. In such *customer branch*, developers might change logos and texts or implement customer-specific features. Changes for specific customers are not merged back into the main branch. In the other direction, we can still merge recent developments and bug fixes from the main branch into the customer branches, if desired. Staples and Hill (2004) describe the experience with this a-branch-per-product style of product line development.

An obvious drawback from customer branches (or product branches) is that each customer that requests a variation needs an own branch with custom modifications. Instead of feature-oriented developing where features can be combined flexibly, we develop custom products.

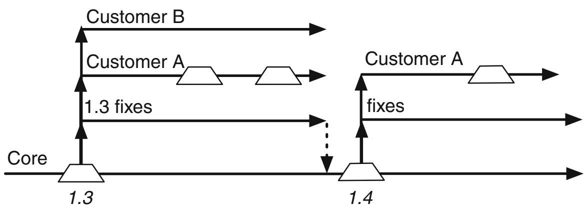


Fig. 5.3 Branching for customer-specific variations: a branch per product (adopted from Staples and Hill 2004)

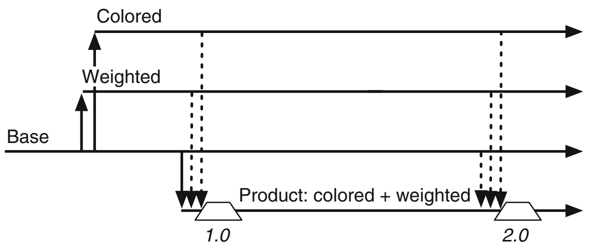


Fig. 5.4 Developing product lines by merging per-feature branches

Instead of developing each product in a separate branch, we can also implement each feature in a distinct branch and create products by merging corresponding feature branches. We exemplify this pattern for the graph example in Fig. 5.4: Developers create a branch per feature (Colored and Weighted in this case) and implement that feature in the branch. To create a product for a given feature selection, a developer merges all changes from all relevant feature branches (for Release `1.0`). If changes occur in individual branches, they can be merged again into the product branch (shown for Release `2.0`). Note that each feature branch also contains the base code, not only the code of the feature; how to trace a feature to its implementation is no longer obvious (see *feature traceability* in Sect. 3.2.2, p. 54).

5.1.3 Discussion

Developers use version-control systems routinely in their work. Since developers use them to track revisions anyway, it seems tempting to use the well-known branching and merging techniques and mature tools for feature variability. In contrast to language-based variability, branching can be used *uniformly* for code and noncode artifacts (such as models, documentation, build scripts, license files, and binary files; see *uniformity* Sect. 3.2.6, p. 60), and changes can be applied at *arbitrary granularity* (from removing directories to changing individual characters; see *granularity* in Sect. 3.2.5, p. 59). Also changes that crosscut an implementation are straightforward, as developers can simply invasively change arbitrary code fragments in the entire code base without upfront preplanning (see *crosscutting features* and *preplanning* Sects. 3.2.3 and 3.2.1, p. 55 and 53). Especially, in the beginning of a project, making a quick change for a customer in a distinct branch is easy without extensive preplanning. Developers simply create a branch and apply changes, instead of thinking about patterns, extensions points, or information hiding. Often, branches are complemented with other implementation techniques in later phases of the project.

However, as a product line evolves, problems accumulate. Unless only few small variations are required for few customers, the use of version-control systems should be restricted to revision control. Version-control systems should not be used for feature variability; its drawbacks far outweigh its benefits. Here are some reasons:

- Version-control systems encourage the development of distinct products of the product line, not features. Instead of developing individual features in our graph example, we would rather develop a directed graph in one branch and an undirected graph in another branch and subsequently open another branch if we need weights. In principle, we could develop each feature in a separate branch and merge them depending on a feature selection (as sketched in Fig. 5.4); however, this pattern relies strongly on effective merging and is problematic for related features that touch similar code fragments (see *code tangling* in Sect. 3.2.3, p. 55).
- If there are many customer-specific variants (or many features in distinct branches), evolution becomes difficult. When applying a fix to a code base, the fix must be merged into all (relevant) branches. It is easy to forget such merges and lose track of branches, so branches easily diverge more than intended. Except for determining deltas between branches, there is no effective means to trace features to their implementation (see *feature traceability* in Sect. 3.2.2, p. 54). Branching provides no form of structured reuse or modularity; branches are essentially copies. The merge operation copies changes from one branch into another.
- Finally, merging branches is a problematic composition technique. Although language-specific merge mechanisms exist (Mens 2002; Apel et al. 2012b), merging is usually performed in a text-based fashion, oblivious to the meaning of the merged artifacts. Differences are usually determined in terms of textual differences between lines of text. Those differences are copied as textual *patches*. When the same line has been changed in both branches, merging tools report conflicts. Usually, developers have to resolve conflicts manually. Even worse, as merge tools are based on heuristics, they may miss conflicts and produce incorrect code. In practice, the heuristics of merge tools work well when distinct code fragments have been changed, but may require high manual effort when related or similar changes have been performed in multiple branches. Furthermore, with text-based tools, we give up the potential of detecting language-specific composition problems.

Summary version-control systems
Strong points:

- Well-known, established, and mature tools.
- External infrastructure.
- Arbitrary compile-time customization independent of granularity and cross-cutting (see Sects. 3.1.1, 3.2.3, and 3.2.5, p. 48, 55 and 59).
- Uniform application to source code and noncode artifacts (see Sect. 3.2.6, p. 60).
- Only minor effort for preplanning.

Weak points:

- Mixture of revisions and variants.
- Encourages development of variants, not features (inconvenient encoding of feature-oriented development). No feature traceability, no separation of feature code, and no information hiding (see Sects. 3.2.2–3.2.4, p. 54–57).
- No structured reuse (copy and edit of plain text).
- Relies on merging, prone to conflicts and errors.
- Hard to maintain with many branches. Fixes must be merged potentially into many branches.
- Almost all practical tools are text based.

5.2 Build Systems

A *build system* is responsible for scheduling and executing all build-related tasks, which may include running generators, compiling source code, running tests, and creating and copying deliverable units. With a build system, developers document and automate the build process. Especially in large projects, a build process is not trivial, since many different build tools (compilers, linkers, parser generators, testing frameworks, documentation generators) and dependencies (libraries, tools) are involved. Like version-control systems, build systems are a part of *software configuration management*.

There are many different build systems with different levels of sophistication. In the simplest case, a build system is a shell script that executes the relevant tools with corresponding parameters. More sophisticated systems, such as *make*, *ant*, and *maven*, support multiple build targets, manage dependencies, avoid unnecessary recompilation with incremental builds, download and update required libraries and tools automatically, and create build reports.

Since a build system already decides when, what, and how to compile, it is an obvious candidate to manage compile-time variability (see *binding times* in Sect. 3.1.1, p. 48). Variability can be encoded in many different ways. In the following, we illustrate three common approaches.

5.2.1 Variability in Build Scripts

Shell scripts and batch files are typical means to automate tasks. As such, they can also be used to automate builds. A simple build script in shown in Fig. 5.5 (left): The

```
#!/bin/bash -e

rm *.class
javac Graph.java Edge.java Node.java \
      Color.java
jar cvf graph.jar *.class
```

```
#!/bin/bash -e

if test "$1" = "--withColor"; then
  cp Edge_withColor.java Edge.java
  cp Node_withColor.java Node.java
else
  cp Edge_withoutColor.java Edge.java
  cp Node_withoutColor.java Node.java
fi

rm *.class
javac Graph.java Edge.java Node.java
if test "$1" = "--withColor"; then
  javac Color.java
fi

jar cvf graph.jar *.class
```

Fig. 5.5 A build script for the graph example without variability (*left*) and with variability (*right*)

script first removes all old class files, subsequently compiles all Java files of the graph example, and finally packages all resulting class files in a JAR file for distribution.

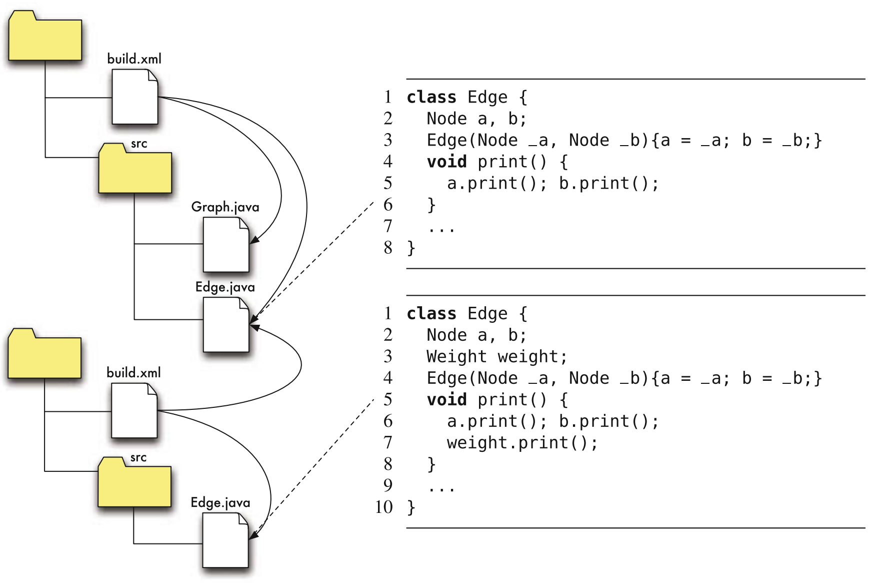
Now, let us introduce variability into this build script. In Fig. 5.5 (right), we check whether the parameter withColor is provided, and we compile one of two implementations of the classes Node and Edge, one with and one without colors. Further, we compile class Color conditionally. In this example, we use the parameter approach from Sect. 4.1 and apply it at build-system level. Depending on the parameters, different files are compiled and packaged.

Instead of passing configuration options to the build script as command-line parameters, build systems can read configuration options also from configuration files (potentially generated by a feature-selection tool). Furthermore, build systems can determine configuration options automatically by inspecting the current context; for example, they can read the operating system's localization settings or detect whether certain hardware features or software libraries are available.

When building against external libraries, variability in a build script can control in which libraries (and which library revision and variant) the product is compiled with (van der Storm 2004). Finally, features can control *how* files are compiled, including triggering optimizations and including debugging information.

5.2.2 Custom Build Scripts

Staples and Hill (2004) describe a setup in which product line developers create a custom build script for each customer (that is, for each product of the product line). They store a base implementation in one directory and customer-specific build scripts together with corresponding customer-specific extensions in other directories. A customer-specific build script may replace files from the base implementation and can add additional files to the build (see Fig. 5.6).



```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b){a = _a; b = _b;}
  void print() {
    a.print(); b.print();
  }
  ...
}
```

```
class Edge {
  Node a, b;
  Weight weight;
  Edge(Node _a, Node _b){a = _a; b = _b;}
  void print() {
    a.print(); b.print();
    weight.print();
  }
  ...
}
```

Fig. 5.6 Customer-specific extensions encoded with build-system variability (adopted from Staples and Hill 2004)

In principle, this encoding is similar to branching in version control (see Sect. 5.1). However, files that are not changed are not copied. When changing a file of the base implementation, the change is merged into configurations that replace the file.

5.2.3 Case Study: Build-System Variability in Linux

The Linux kernel is built using a custom build system, *Kbuild*, which comprises a hierarchy of build scripts as input for the tool *make*, following a specific convention. The build scripts decide which of the several thousand C files should be compiled and linked together when building a Linux kernel. The logic for selecting source files for a particular configuration is spread over more than 600 smaller build scripts in the entire source tree (Berger et al. 2010a; Dietrich et al. 2012b). Some build scripts are only conditionally executed depending on the feature selection.

Typical lines of a build script look like this: "obj-y += foo.o" means that file foo.c should be compiled and linked into the kernel. Similarly, the line "obj-m += foo.o" specifies that file foo.c should be built as a loadable kernel module and "lib-y += foo.o" means that the file should be included as a library. Variability is encoded in the form "obj-(CONFIG_FOO) += foo.o", in which CONFIG_FOO is a feature name selectable in Linux's feature model. During configuration (see *requirements analysis* in Sect. 2.2.2, p. 24), CONFIG_FOO is set either to y (compile), m (compile as module),

```
#
# Makefile for the video capture/playback device drivers.
#

tuner-objs      :=      tuner-core.o

videodev-objs   :=      v4l2-dev.o v4l2-ioctl.o v4l2-device.o

obj-$(CONFIG_VIDEO_DEV) += videodev.o v4l2-int-device.o
ifeq ($(CONFIG_COMPAT),y)
  obj-$(CONFIG_VIDEO_DEV) += v4l2-compat-ioctl32.o
endif

obj-$(CONFIG_VIDEO_V4L2_COMMON) += v4l2-common.o

ifeq ($(CONFIG_VIDEO_V4L1_COMPAT),y)
  obj-$(CONFIG_VIDEO_DEV) += v4l1-compat.o
endif

obj-$(CONFIG_VIDEO_TUNER) += tuner.o
obj-$(CONFIG_VIDEO_TVAUDIO) += tvaudio.o
obj-$(CONFIG_VIDEO_TDA7432) += tda7432.o
obj-$(CONFIG_VIDEO_TDA9875) += tda9875.o

...

EXTRA_CFLAGS += -Idrivers/media/common/tuners
```

Fig. 5.7 Excerpt of the build script of the Linux kernel (*drivers/media/video/Makefile*)

or n (do not compile). This way, *Kbuild* controls which files are included for compilation for a given feature selection and how they are compiled. Of 9146 C files in Linux release 2.6.33.3, 97 % are included conditionally in some feature selections.

Furthermore, several patterns exist for libraries and for grouping files that belong to the same feature. Compiler parameters, such as include paths and optimization level, can be set depending on configuration options from the feature model for all files as well as for individual files. As *Kbuild* relies on a Turing-complete language, complex conditions can be encoded. Figure 5.7 shows an excerpt of the build script responsible for selecting video drivers.

In the Linux kernel, the build system controls variability at the file level: It selects which files are passed to the compiler. Variability at finer granularity is expressed using preprocessor directives inside files, which we discuss in Sect. 5.3.

5.2.4 Discussion

Large software projects use a build system that controls which files are compiled and how. It is natural to use the build system for variability at compile time. Typically, build systems use a parameter-based approach (as discussed in Sect. 4.1) when they are executed at compile time (see *binding times* in Sect. 3.1.1, p. 48).

Build systems are suitable to control variability at the file level (see *granularity* in Sect. 4.4, p. 89). They can directly control which files to compile under which condition. Therefore, we typically consider them as an annotation-based mechanism at the file level (see *annotation versus composition* in Sect. 3.1.3, p. 50). Typical encodings support choosing from alternative variants of a file or replace files with customer-specific variants. At the file level, a build system is language-agnostic and can uniformly implement variability for different code and noncode artifacts (see *uniformity* in Sect. 3.2.6, p. 60). As such, arbitrary changes are possible at compile time, there is no need for extensive preplanning activities, but there is no notion of consistency or modularity.

As long as features can be mapped to files, build systems are well-suited for feature-oriented product lines (see *feature traceability* in Sect. 3.2.2, p. 54). File-level granularity can be problematic though. In case of only small changes, we need to replicate entire files. For example, in Fig. 5.6, we had to copy the entire class implementation to add one field weight and one print statement. When multiple features affect the same file, we need mechanisms beyond file selection and file replacement. However, even with file-level variability, many implementation patterns can be used to reduce replication and enable composition. For example, developers can encode feature-specific extensions as observers, subclasses, strategy objects, and decorators in separate files (see *design patterns* in Sect. 4.2, p. 69) or use the factory-method pattern (Gamma et al. 1995) to control which classes to instantiate in a central place.

Beyond file-level variability, a build system typically orchestrates other variability mechanisms. For example, it can conditionally apply textual patches, load a file from a selected branch in a version-control system (see Sect. 5.1, p. 99), run preprocessors (see Sect. 5.3, p. 110), or create a suitable configuration file for configuration parameters (see Sect. 4.1, p. 66), or plug-in loaders (see Sect. 4.3, p. 79).

Many build systems are constructed with Turing-complete scripting languages and allow sophisticated encodings of variability. Therefore, when trying to *statically analyze* build scripts (for example, to extract traceability links between features and files), issues similar to those of the parameter approach arise (see Sect. 4.1, p. 66): Most static analysis tasks on build systems are undecidable; the build system may just call arbitrary shell scripts. A precise and complete analysis is only possible for restricted languages or when developers use only specific patterns. For example, several researchers have tried to analyze the build system of the Linux kernel *Kbuild* to extract traceability links between features and files (Berger et al. 2010a; Nadi and Holt 2010; Dietrich et al. 2012a). The example in Fig. 5.7 is still rather simple, but also not trivial with ifqe statements evaluated at compile time. Due to the power of the underlying scripting language, all three approaches had to approximate or simulate build-system behavior. We return to this problem when discussing product line analysis in Sect. 10.2.3.

We recommend (a) to stick to simple variability patterns when using a build system (or even restricted statically analyzable languages) (b) to always complement build systems with other variability mechanisms at subfile level, and (c) never clone code for file-level variability. In combination with other variability mechanisms, build systems are a common, suitable, and well-known means to implement variability, especially as a form of conditional compilation for entire files.

Summary build systems
Strong points:

- Well-known and established tools.
- Suitable for conditional compilation at file level (see Sect. 3.2.5, p. 59).
- Orchestration of other variability mechanisms.
- Arbitrary compile-time customization (see Sects. 3.1.1 and 3.2.5, p. 48 and 59).
- Uniform application to source code and noncode artifacts (see Sect. 3.2.6, p. 60).
- No extensive preplanning necessary.

Weak points:

- Limitation to file-level variability and discouragement of systematic preplanning leads to code replication.
- Largely unsuitable for feature-oriented programming at a fine grain (see Sect. 3.2.5, p. 59).
- No notion of modularity (see Sects. 3.2.3 and 3.2.4, p. 55 and 57).
- Feature traceability at the file level only (see Sect. 3.2.2, p. 54).
- Undisciplined and complex scripts can become hard to maintain and analyze (see Sect. 10.2.3, p. 257).

5.3 Preprocessors

A *preprocessor* is a tool that manipulates source code before compilation. A popular preprocessor is the C preprocessor *cpp*, which is used in almost every C and C++ project. It provides directives to inline files, to define macros, and to remove code fragments based on user-defined conditions. In addition to *cpp*, many other preprocessors exist and are used for specific purposes. Most important in product line development, preprocessors typically provide facilities for conditional compilation, where marked code fragments in the source code are conditionally removed before compilation—`#ifdef` and `#endif` in *cpp*. Conditional compilation is probably the most common mechanism for implementing variability in product lines in industrial practice.

5.3.1 *The C Preprocessor* `cpp`

Historically, *cpp* has been developed for lightweight metaprogramming. It provides facilities for file inclusion, lexical macros, and conditional compilation. At `#include` directives, the preprocessor lexically inlines files, for example, to reuse header files. Lexical macros are defined with #define directives. After defining a macro, the preprocessor replaces all occurrences of a corresponding token by another token sequence. Macros serve as compile-time variables and functions; they can be defined, redefined, and undefined while preprocessing a file. Finally, conditional compilation, with #if, #ifdef, and similar directives can suppress token sequences so that they are not compiled. The #if directives evaluate compile-time expressions, typically checking whether macros are defined or not.

The C preprocessor is oblivious to the underlying language and operates on token sequences. Hence, it can and has been used on other languages with a similar lexical syntax than C, such as C++, Fortran, Java, and assembly languages. Furthermore, many language environments provide their own but similar preprocessing facilities, including C#, Visual Basic, Erlang, Pascal, D, and PL/SQL, sometimes in separate tools, sometimes built into the compiler. For languages without such facilities, developers have implemented their own preprocessors, such as *Munge*[2] and *Antenna*[3] for Java.

5.3.2 Implementing Variability with Preprocessors

The typical way to implement variability with preprocessors is to wrap a code fragment with conditional-compilation directives, such as #ifdef and #endif. We say such code fragment is *annotated* (see *annotation versus composition* in Sect. 3.1.3, p. 50). Depending on whether certain macros are defined (potentially using command-line parameters passed to the preprocessor), the code fragments are included or removed before compilation. To implement feature-oriented software product lines, we reserve a name per feature (typically F as the name of the feature), and define the macro (using "#define F") only when the feature is selected.

In Fig. 5.8, we show an example of conditional compilation with the C preprocessor in a code excerpt from Oracle's Berkeley DB. In addition to the common #ifdef and #endif directives, we can see negation (#ifndef), alternatives (#else), and nesting of preprocessor directives. The example illustrates also the common case of long annotated code fragments, here annotating over 100 lines of code (see comment in Line 16). The preprocessor removes code according to macro definitions before compilation.

For completeness, in Fig. 5.9, we show an implementation of our graph product line implemented with conditional compilation. Since the example is implemented in Java, we use the preprocessor *Munge* instead of *cpp*. The syntax is similar: code fragments are annotated by feature directives using IF and END inside comments (so they do not interfere with development environments such as Eclipse). We run the preprocessor *Munge* as follows:

[2] http://weblogs.java.net/blog/tball/archive/munge/doc/Munge.html

[3] http://antenna.sf.net

```
static int __rep_queue_filedone(dbenv, rep, rfp)
  DB_ENV *dbenv;
  REP *rep;
  __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
  COMPQUIET(rep, NULL);
  COMPQUIET(rfp, NULL);
  return __db_no_queue_am(dbenv);
#else
  db_pgno_t first, last;
  u_int32_t flags;
  int empty, ret, t_ret;
#ifdef DIAGNOSTIC
  DB_MSGBUF mb;
#endif
  // over 100 lines of additional code
#endif
}
```

Fig. 5.8 Excerpt from Oracle's Berkeley DB implementing variability with conditional compilation using *Munge*

```
class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n,m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    /*IF[FEAT_WEIGHTED]*/
    e.weight = new Weight();
    /*END[FEAT_WEIGHTED]*/
    return e;
  }
  /*IF[FEAT_WEIGHTED]*/
  Edge add(Node n, Node m, Weight w) {
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  /*END[FEAT_WEIGHTED]*/
  void print() {
    for(int i=0; i<edges.size(); i++){
      ((Edge) edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}


/*IF[FEAT_COLORED]*/
class Color {
  static void setDisplayColor(Color c)...
}
/*END[FEAT_COLORED]*/
class Node {
  int id = 0;
  /*IF[FEAT_COLORED]*/
  Color color = new Color();
  /*END[FEAT_COLORED]*/
  Node (int _id) { id = _id; }
  void print() {
    /*IF[FEAT_COLORED]*/
    Color.setDisplayColor(color);
    /*END[FEAT_COLORED]*/
    System.out.print(id);
  }
}

class Edge {
  Node a, b;
  /*IF[FEAT_COLORED]*/
  Color color = new Color();
  /*END[FEAT_COLORED]*/
  /*IF[FEAT_WEIGHTED]*/
  Weight weight;
  /*END[FEAT_WEIGHTED]*/
  Edge(Node _a, Node _b) {a=_a; b=_b;}
  void print() {
    /*IF[FEAT_COLORED]*/
    Color.setDisplayColor(color);
    /*END[FEAT_COLORED]*/
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
    /*IF[FEAT_WEIGHTED]*/
    weight.print();
    /*END[FEAT_WEIGHTED]*/
  }
}

/*IF[FEAT_WEIGHTED]*/
class Weight {
  void print() { ... }
}
/*END[FEAT_WEIGHTED]*/
```

Fig. 5.9 Graph library: Variability implemented with conditional compilation

```
#ifdef FEAT_BIGINT
  #define SIZE 64
#else
  #define SIZE 32
#endif

... allocate(SIZE) ...;
```

```
#ifdef FEAT_WINDOWS
  #include <windows.h>
#else
  #include <unix.h>
#endif

... fopen(...) ...;
```

```
#ifdef FEAT_SELINUX
  #define FEAT_LINUX 1
  #undef  FEAT_WINDOWS
#endif

#ifdef FEAT_WINDOWS
...
```

```
#ifdef FEAT_RAND
int rand() { ... }
#else
#define rand(...) 0
#endif

int i = 3 + rand();
```

Fig. 5.10 Further preprocessor usage pattern for variability implementation: Alternative macro definitions (top left), alternative includes (top right), dependent feature definitions (bottom left), and alternative function definition (bottom right)

```
java Munge -DFEAT_WEIGHTED -DFEAT_COLORED
Graph.java Node. java ... targetDirectory
```

in which the -D parameters define macros. *Munge* removes all annotated code fragments, for which the corresponding macros have not been defined. Subsequently, we call the compiler on the preprocessed code in the target directory.

Of course, developers can also use other preprocessor mechanisms to encode variability, typically in combination with conditional compilation. Four common patterns are illustrated in Fig. 5.10. First, we can define a macro differently, depending on the feature selection. In our example, SIZE expands to `64` or `32` depending on whether feature BigInt is selected. Second, we can include alternative files, depending on the feature selection. In our example, we include the header files of either Windows or Unix. Third, we can define, redefine, and undefine macros during the preprocessor's execution, among others to encode dependencies. In our example, feature Windows is automatically undefined when feature SELinux is defined, thus potentially overriding feature selections by users. Finally, a pattern that is quite common in the Linux kernel is to provide an alternative to an optional function as a macro (or an empty function definition to be inlined). If feature Rand is not selected in our example, the function call is replaced by a literal '`0`' to be optimized away by the compiler.

5.3.3 *Further Preprocessors*

Annotating and conditionally removing code fragments is a very general concept that takes many different shapes in practice. In this section, we briefly outline some specialized preprocessors to give an overview of the variety of different possibilities.

Java. Natively, Java has limited facilities for conditional compilation. If the compiler can statically determine that the condition of an if statement always evaluates to false, it may chose not to compile the body of the if statement. The Java language spec-

```
class Example {
  public static final boolean DEBUG = false;
  void main() {
    ...
    if (DEBUG)
      System.out.println("debug info");
  }
}
```

Fig. 5.11 Native conditional compilation in Java

ification explicitly provides an exception to detecting unreachable code in the body of if statements, to allow this simple form of conditional compilation at expression level (Gosling et al. 2005, Sect. 14.21). Hence, although using normal parameters, the code in Fig. 5.11 can be seen as a form of conditional compilation. However, not all compilers implement this optimization. By using the same construct, compile-time and run-time variability (see Sect. 3.1.1, p. 48) become difficult to distinguish. Furthermore, conditional compilation is not natively available for removing entire methods or classes; for that, external tools are required.

Antenna. *Antenna* is a preprocessor developed for Java ME projects, which usually target embedded devices with limited resources and varying capabilities. *Antenna* is unusual as it integrates into existing development processes and code editors for Java. It mimics the syntax of the C preprocessor, but, as with *Munge*, preprocessor directives are written in comments not to interfere with existing code editors. Instead of removing code fragments, *Antenna* comments out code fragments annotated with deselected features and uncomments code fragments of selected features. This way, developers can edit Java code in their normal code editors and quickly switch between alternative feature selections; code is changed in-place and not generated in separate files. There are even plug-ins to integrate *Antenna* and the switching between feature selections into development environments such as NetBeans and Eclipse.

Customizable preprocessors. Although the C preprocessor can be used on all languages with a token structure similar to C, its fixed notation can be limiting: Preprocessor directives always start with # in a new line. There are several customizable preprocessors that developers can adjust preprocessor syntax to align with the host language's syntax, for example, to appear in comments in the respective language. Examples are *GPP*—Generic Preprocessor, *GNU M4*, and preprocessors provided as part of the commercial product line tools *pure::variants* and *Gears*.

Syntactic preprocessors. Most preprocessors work at the lexical level: They transform token sequences without considering the underlying language structure. Several syntactic preprocessors exist that take a specific language into account. For example, consider the code excerpt in Fig. 5.12 in the tag-and-prune approach of Boucher et al. (2010). There is no end tag; instead, an annotation applies to a whole syntactic unit, such as a function, a statement, or a case of a switch statement. Syntactic preprocessors promise to be less prone to subtle syntax errors, as they enforce a certain discipline of annotations. Instead of removing token sequences, a syntactic preprocessor performs transformations as rewrites of abstract syntax trees. In addi-

```
/*@feature:RECV_MIN@*//*@!file_feature!@*/
(...)
void cfdp_receiver_handle_PDU(cfdp_receiver* const me, struct cfdp_buffer* PDU_buffer,
     CFDP_PDU_type_t PDU_type) {
  {
    /*@feature:RECV_INACTIVITY@*/
    /* Restart inactivity timer */
    cfdp_timer_start(&(me->timer_inactivity),me->config.timeout_inactivity);

    /* Handle PDU and dispatch it depending on its type */
    switch (PDU_type)
    {
      /*@feature:RECV_MIN_ACK@*/
      case CFDP_PDU_ACK_FINISHED:
      {
        cfdp_receiver_handle_PDU_eof_no_error(me,PDU_buffer);
      }
      break;
      case CFDP_PDU_EOF_NO_ERROR:
      {
        cfdp_receiver_handle_PDU_eof_no_error(me,PDU_buffer);
      }
      break;
    }
    (...)
  }
}
```

Fig. 5.12 Syntactic preprocessor in the *tag-and-prune* approach of Boucher et al. (2010)

tion to conditional compilation, syntactic preprocessors typically (or even primarily) provide syntactic macros that rewrite language structures instead of replacing tokens (Brabrand and Schwartzbach 2002). Syntactic preprocessors and, especially, syntactic macro systems are well known from Lisp-like languages and C++ templates (Brabrand and Schwartzbach 2002), but have been also explored with a look and feel familiar to C programmers (Weise and Crew 1993; McCloskey and Brewer 2005; Kästner et al. 2009b; Boucher et al. 2010; Batory et al. 2011).

Preprocessors for nontext artifacts. Many preprocessors can be uniformly applied to all kinds of textual artifacts. However, preprocessor annotations have been explored also for graphical models and other nontext artifacts. For graphical models in Eclipse, both *fmp2rsm* (Czarnecki and Antkiewicz 2005) and *FeatureMapper* (Heidenreich et al. 2008b) provide annotation and preprocessing facilities. The tool *fmp2rsm* uses UML stereotypes to encode annotations, whereas *FeatureMapper* uses an external syntactic mapping. Both tools provide some form of syntactic macro facilities. In Fig. 5.13, we show a screenshot of an annotated entity-relationship model in *FeatureMapper*. Similarly, specialized preprocessors for documents in word processors are provided by vendors of the commercial product line tools *pure::variants* and *Gears* and have been explored in academic contexts as well (Batory et al. 2011).

Template processors. Most preprocessors discussed so far were rather restricted to replacing or removing tokens. There are several preprocessors that allow you to write powerful generators. Some provide Turing-complete facilities beyond conditional compilation. Typical features of template engines not provided by preprocessors are loops over some data structures and arbitrary computations of val-

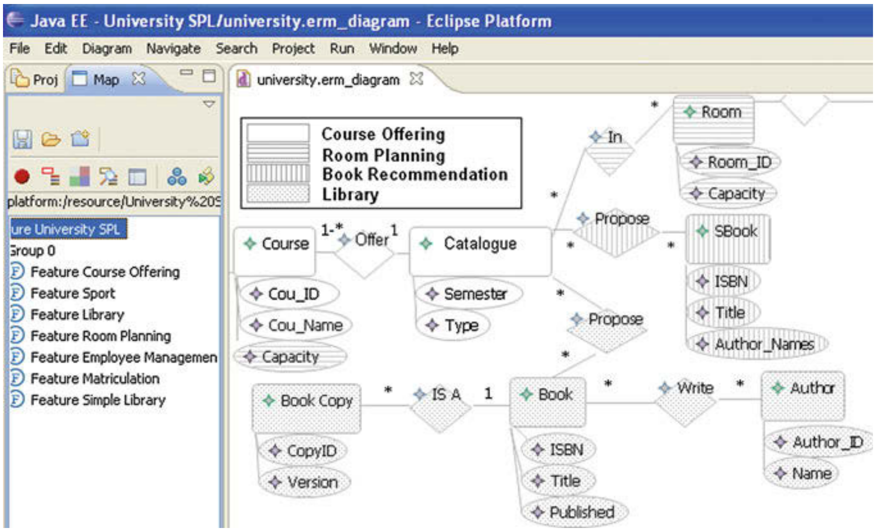


Fig. 5.13 Model annotations of an entity-relationship diagram in *FeatureMapper* (Siegmund et al. 2009a)

ues at generation time. *XVCL* by Jarzabek et al. (2003) is a lexical template engine based on frame technology that was explicitly designed for product line development. Another example of a lexical template engine is *StringTemplate* by Parr (2004). Both are applicable to any text-based language; but, like lexical macros, they are oblivious to the underlying language structure. Several syntactic template engines exist for specific languages that can provide guarantees about the generated code, for example, that it is always syntactically correct or well-typed (Huang et al. 2005; Arnoldus et al. 2007; Huang and Smaragdakis 2011).

5.3.4 Disciplined Annotations

Preprocessors are powerful tools that allow almost arbitrary source-code manipulation at compile time. The power comes at the price of writing complex and hard-to-maintain and hard-to-understand source code. Similar to the goto statement, preprocessors have earned a poor reputation.

Preprocessor annotations of lexical preprocessors can align with the underlying structure but do not have to: Developers can use conditional compilation to annotate entire syntactic blocks of source, such as entire functions or entire statements. In this case, annotations *align* with the syntactic structure of the source code. We say an annotation is *disciplined*, if it aligns with the source code structure of a language. Disciplined annotations can be mapped to elements of the documents abstract syntax tree and are restricted to selected structures, typically, top-level declarations, fields,

```
#if defined(__MORPHOS__) &&
     \defined(__libnix__)
extern unsigned long *__stdfiledes;

static unsigned long
    fdtofh(int filedescriptor) {
   return __stdfiledes[filedescriptor];
}
#endif
```

```
void tcl_end() {
#ifdef DYNAMIC_TCL
  if (hTclLib) {
    FreeLibrary(hTclLib);
    hTclLib = NULL;
  }
#endif
}
```

Fig. 5.14 Excerpts of *vim* with disciplined annotations: an annotation around entire top-level declarations (left) and an annotation round an entire statement (right) (Liebig et al. 2011)

```
1   int n = NUM2INT(num);
2 #ifndef FEAT_WINDOWS
3   w = curwin;
4 #else
5   for (w = firstwin; w != NULL;
6       w = w->w_next, --n)
7 #endif
8     if (n == 0)
9       return window_new(w);
```

```
  if (!ruby_initialized) {
#ifdef DYNAMIC_RUBY
    if (ruby_enabled(TRUE))
#endif
      ruby_init();
```

```
int put_eol(fd)
    FILE *fd;
{
  if (
#ifdef USE_CRNL
    (
#ifdef MKSESSION_NL
      !mksession_nl &&
#endif
      (putc('\r', fc) < 0)) ||
#endif
      (putc('\n', fd) < 0))
    return FAIL;
  return OK;
}
```

Fig. 5.15 Excerpts of *vim* with undisciplined annotations at substatement and subexpression level (Liebig et al. 2011)

and statements, but not subexpressions or parameter declarations. In Figs. 5.14 and 5.15, we show a number of code fragments from the source code of the text editor *vim* that we judge as disciplined or undisciplined, respectively. Notice how the annotation in the first example of Fig. 5.15 changes the structure and decides whether Line 8 is in the body of a for loop or not.

The notion of disciplined annotations can help developers to avoid complex ad hoc implementations that might be hard to maintain. We argue that disciplined annotations are usually also easier to read and maintain. Baxter and Mehlich (2001) observed about undisciplined annotations: "The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source." Undisciplined annotations are not only problematic for developers, but also for tools: Even parsing code with undisciplined annotations is a challenge, so refactoring engines rarely fully support code with undisciplined annotations. Refactoring engines for Java are more advanced than those for C and C++, because in the latter languages the preprocessor hinders proper parsing, analysis, and rewrites.

So far, few tools enforce disciplined annotations (usually only syntactic preprocessors, discussed above). Typically, developers agree upon them by convention. For example, the Linux developers have agreed upon a convention to restrict the use of conditional compilation to few disciplined patterns: “Code cluttered with ifdefs is difficult to read and maintain. Do not do it. Instead, put your ifdefs in a header, and conditionally define static inline functions, or macros, which are used in the code”.[4]

In an empirical study of typical C code, Liebig et al. (2011) found that most #ifdef annotations (84 %) are already in a disciplined form. The remaining annotations can mostly be rewritten if desired. The study also discusses benefits and limitations of disciplined annotations in more detail.

5.3.5 Preprocessors in Practice

Preprocessors are widely used in practice, especially, in C and C++ projects, to a much lesser degree in Java projects. There are several studies on preprocessor usage in open-source projects, for example, by Ernst et al. (2002) and Liebig et al. (2010), (2011). Here we summarize some key results from Liebig’s work, which focused primarily on conditional compilation and product-line development.

By looking at 40 C open-source projects from different domains, Liebig et al. (2010, 2011) found that *all* analyzed open-source projects contained some amount of conditional compilation with between 13 and 16 167 features covering between 2.5 % and 69 % of all C code. Over all projects, on average, each feature contributed 7 #ifdef blocks with a total of 60 lines of code. Only about 10 % of all features inject the same code in multiple locations (also known as homogeneous extensions, see later discussions in Sect. 6.3.1, p. 153), all other code fragments inject always different code at each extended location.

In practice, most uses of #ifdef directives are disciplined and at a comparably coarse granularity (see *granularity* in Sect. 3.2.5, p. 59). On average, only 1.8 % of all #ifdef blocks modify code inside a statement. Most #ifdef blocks actually guard entire statements (52 %, on average) or entire functions or other top-level declarations (46 %, on average). Of these #ifdef blocks, 84 % are disciplined, according to our notion of disciplined.

By comparing the projects, we can also find that the number of features and the amount of #ifdef-guarded code grows roughly linearly with the size of the project (measured in lines of code), whereas the scattering and complexity of #ifdef differs strongly between projects but is independent of the size of the project.

All of these values differ quite significantly between individual projects, but are consistent in their general tendency: conditional compilation is used intensively in open-source C projects.

In Table 5.1, we list some results of these studies for all 40 analyzed projects. For more data and more information about the metrics and studies, refer to the original publications (Liebig et al. 2010, 2011).

[4] see */Documentation/SubmittingPatches* in the kernel sources.

Table 5.1 Preprocessor usage in 40 open-source C systems (data excerpt from Liebig et al. (2010, 2011))

Project	LoC	Feat.	FeatC (in %)	LoC/Feat.	Scat.	Granularity (in %) Global	Stmt.	Finer	Discipl. (in %)
apache	208 785	1 158	22	39	5.6	39	59	1.8	80.5
berkeley-db	185 433	915	15	31	4.7	31	68	0.8	88.0
cherokee	50 571	328	15	23	3.7	52	48	0.7	77.4
clamav	74 142	285	15	38	5.9	52	48	0.2	82.6
dia	128 400	91	4	59	5.4	59	40	1.6	88.1
emacs	232 224	1 373	33	55	10.6	46	52	2.2	87.4
freebsd	5 845 346	16 167	14	52	10.5	48	51	1.6	84.7
gcc	1 599 365	5 063	18	56	7.0	56	41	3.4	82.2
ghostscript	438 379	816	5	27	4.0	53	46	1.2	89.5
gimp	585 875	392	3	49	6.5	56	44	0.1	90.5
glibc	729 065	3 012	12	29	7.3	60	38	1.9	80.2
gnumeric	253 280	291	5	41	3.2	19	76	5.2	83.6
gnuplot	74 801	434	27	47	7.7	44	53	2.9	79.1
irssi	49 450	55	3	23	2.5	38	61	0.5	86.0
libxml2	205 618	2 047	68	68	8.4	70	29	0.3	93.5
lighttpd	38 499	167	22	51	4.7	32	60	7.5	91.3
linux	5 882 780	9 102	11	71	6.1	52	46	1.2	92.3
lynx	115 238	806	38	54	11.6	31	67	2.0	79.5
minix	62 573	356	17	30	5.1	53	47	0.3	92.5
mplayer	588 625	1 236	19	92	5.8	38	60	2.2	82.8
mpsolve	10 142	13	3	20	2.5	59	41	0.0	100.0
openldap	243 691	708	27	95	4.7	35	62	2.6	87.9
opensolaris	8 107 717	10 901	20	151	10.4	39	60	1.0	75.7
openvpn	33 777	276	69	84	6.7	43	53	3.8	92.5
parrot	101 482	539	26	49	6.1	62	38	0.1	89.9
php	565 046	2 426	35	81	5.6	50	49	1.3	82.9
pidgin	267 349	576	15	70	3.2	42	56	1.9	86.2
postgresql	448 495	692	5	34	4.9	42	57	0.9	80.1
privoxy	23 597	153	38	59	7.0	39	59	1.8	74.2
python	370 119	5 127	27	20	4.1	23	76	1.4	89.5
sendmail	81 158	880	39	36	5.9	39	58	2.7	74.8
sqlite	93 213	292	55	174	7.7	45	54	1.3	88.1
subversion	506 085	409	6	70	14.0	61	38	0.2	69.7
sylpheed	100 326	271	19	70	4.2	56	42	2.2	83.3
tcl	131 284	2 481	20	11	3.8	69	30	0.6	80.9
vim	222 682	779	60	170	19.3	23	69	8.4	72.6
xfig	72 067	107	7	48	4.7	43	55	1.9	80.6
xine-lib	484 898	1 692	35	101	5.3	44	54	1.2	85.2
xorg-server	513 702	1 360	19	70	9.0	48	50	1.4	83.9
xterm	48 481	453	40	42	8.1	42	57	1.2	86.7
Sum/Avg.	29 773 760	74 229	23	60	6.6	46	52	1.8	84.4

LoC = n lines of normalized C code; Feat. = total of n feature constants in conditional-compilation directives; FeatC = n percent of code in the project belonging to any feature; L/Feat. = n lines of code on average per feature; Scat. = average scattering per feature over n annotated code fragments; Granularity = n percent of annotations at global/statement/finer granularity (see text); Discipl. = n percent of annotations in a disciplined form (see text)

5.3.6 Discussion

Preprocessors are controversial. On the one hand, they are used in practice in many product-line projects. On the other, academics heavily criticize preprocessors as summarized in the claim "#ifdef Considered Harmful" (Spencer and Collyer 1992) and in the colloquial term "*#ifdef hell*" (Lohmann et al. 2006a), and recommend that their use should be abandoned. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability (for example, Spencer and Collyer 1992; Krone and Snelting 1994; Favre 1995, 1997; Ernst et al. 2002; Adams et al. 2008b). In the following, we examine the advantages and disadvantages of preprocessors more closely.

Benefits

On the positive side, preprocessors are lightweight and easy-to-use tools for *compile-time* variability that are already available in many language environments (see *binding times* in Sect. 3.1.1, p. 48). They have a *simple programming model: Annotate and conditionally remove code.* Most developers, especially C developers, are familiar with them; otherwise, they are easy to learn. In environments without preprocessors, they can be added as lightweight tools.

Developers can apply lexical preprocessors such as the C preprocessor *uniformly* to artifacts of different kinds, even noncode and nontext artifacts (see *uniformity* in Sect. 3.2.6, p. 60). Thus, the same variability mechanism can be used for source code, grammars, models, documentation, and so forth. They allow changes at arbitrary levels of granularity, not confined by mechanisms of the host language, as illustrated in examples throughout this section (see *granularity* in Sect. 3.2.5, p. 59). Fine-grained compile-time variability allows controlling precisely what code is actually deployed, even at statement level. Even disciplined annotations scale to many levels of granularity.

Since most preprocessors are rather simple, it is usually easy to extract feature-traceability information (see *feature traceability* in Sect. 3.2.2, p. 54). Annotations can refer directly to features. Although feature code may not be localized in one file, a simple search or an analysis tool can statically determine code locations affected by a feature.

Finally, granularity, expressiveness, and their lightweight nature render preprocessors well-suited for *extractive product line adoption* (see *adoption paths* in Sect. 2.4, p. 39). It is easy to add a few preprocessor directives to existing code, to make code fragments optional or encode alternative. In contrast to frameworks, little preplanning is required, as code is changed invasively (see *preplanning* in Sect. 3.2.1, p. 53). This lightweight, easy-to-learn, and easy-to-adopt nature is what makes them so popular in practice.

Criticism

Preprocessors are criticized for neglecting *separation of concerns* (see Sect. 3.2.3, p. 55). Similar to simple parameter approaches (see Sect. 4.1, p. 66), feature code is scattered across the code base and tangled with code of other features. There is no support of encapsulation (see *information hiding* in Sect. 3.2.4, p. 57, as achievable, for example, with frameworks and components. Even when feature code is roughly separated into distinct files, scattered annotations remain to inject the feature code into the base code (as scattered method invocations or #include directives).

Scattering not only affects source code, but *configuration knowledge* as well. Configuration parameters can be defined, redefined, or undefined within the source code, as exemplified earlier in Fig. 5.10. Such scattering of configuration knowledge can make it hard to understand when or why a certain code fragment is included in a product.

Favre (1995) and Baxter and Mehlich (2001) argue that preprocessors, due to their simplicity, invite developers to make ad hoc extensions and use "quick and dirty" solutions, instead of restructuring the code. Features are steadily added in a patch-by-patch fashion, but never removed or reflected in the design. Such ad hoc use can lead to implementations that are hard to understand and maintain.

Furthermore, preprocessors are criticized for obfuscating source. Although already a problem of the parameter approach, preprocessors intensify the problem, because annotations are typically not part of the language, but added on top by an external tool, multiple languages are intermixed in the same file. Especially, when annotations are applied at a fine grain, are strongly scattered, or are used in undisciplined ways, it can be difficult to follow the control flow in the host language. The term "*#ifdef hell*" reflects this problem.

The degree of source-code obfuscation depends on how the preprocessor is used. Similar to parameters (see Sect. 4.1, p. 66), a small number of disciplined annotations is often perceived as acceptable. However, if applied in an ad hoc fashion, fine-grained annotations and a large number of annotations can quickly reduce readability. We illustrate code obfuscation in Fig. 5.16 with a small constructed example and, in Fig. 5.17, with an illustration of preprocessor directives in the source code of *Femto OS*, a small real-time operating system.

Preprocessors are also considered to be error prone. Most preprocessors lack proper diagnostic tools. Using annotations to implement optional features can easily introduce errors that can be difficult to detect. If used in an undisciplined fashion at token level, even simple syntax errors, such as not closing a corresponding bracket in some feature selections can sneak in undetected. There is no way to check features in isolation, as one might attempt for modular implementations, such as plug-ins or components.

We illustrate a simple syntax error in the adapted code excerpt from Berkeley DB in Fig. 5.18: The opening bracket in Line 4 is closed in Line 17 only when feature Have_Queue is selected, all other products contain a syntax error. The worst part is that compilers cannot detect such syntax errors, unless the developer (or customer) happens to build a product with a problematic feature combination (without fea-

Fig. 5.16 Java code obfuscated by fine-grained annotations with *cpp* (Kästner et al. 2008a)

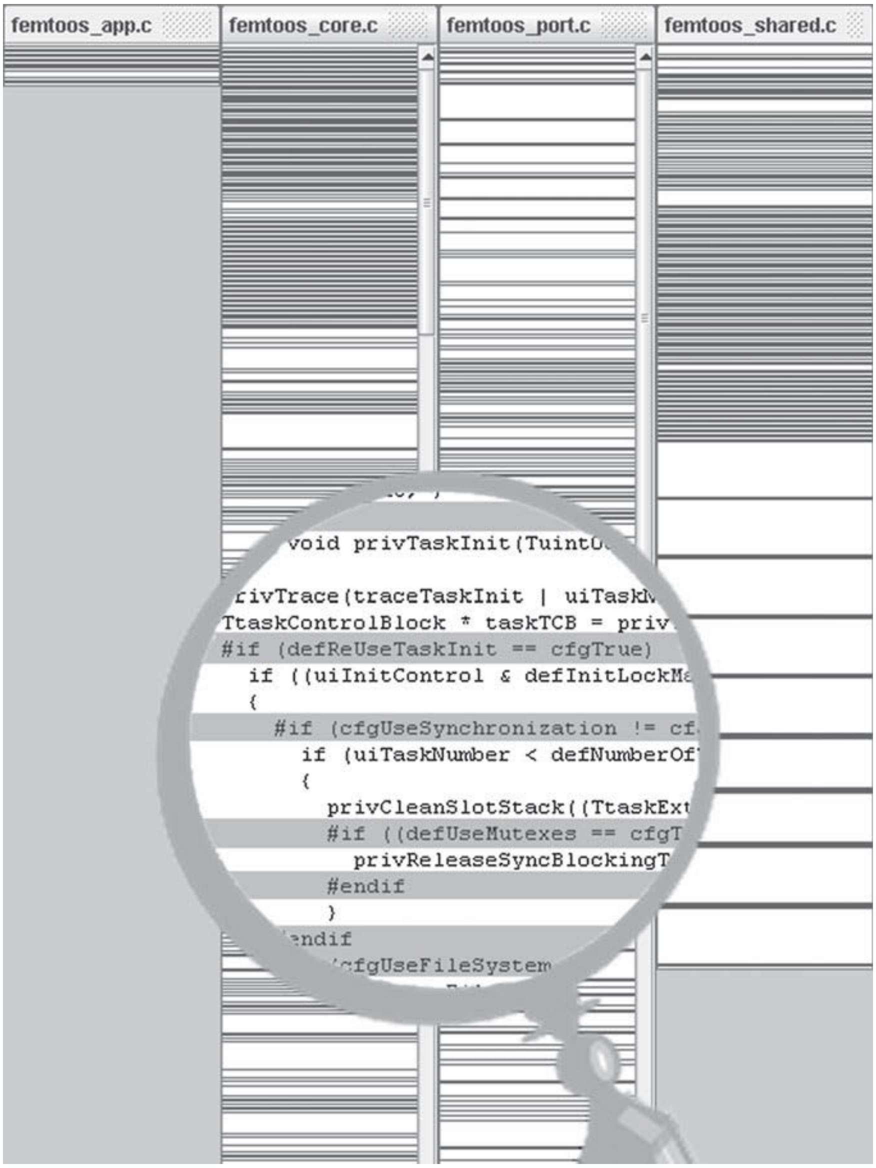
```
class Stack {
  void push(Object o
#ifdef SYNC
  , Transaction txn
#endif
  ) {
    if (o==null
#ifdef SYNC
      || txn==null
#endif
      ) return;
#ifdef SYNC
    Lock l=txn.lock(o);
#endif
    elementData[size++] = o;
#ifdef SYNC
    l.unlock();
#endif
    fireStackChanged();
  }
}
```

ture Have_Queue in our case). However, since there are so many potential products (2^n products for n independent optional features), we might not compile products with a problematic feature combination during initial development. Compiling all products is infeasible due to its high number, so even simple syntax errors might go undetected for a considerable time. The bottom line is that errors are found only late in the development cycle, when they are more expensive to fix. Beyond syntax errors, also type errors and semantic errors can occur, as we discuss in Chap. 10.

Overall, preprocessors are a simple means to implement variability, but one with several dangerous pitfalls. Judging preprocessors as beneficial or problematic depends significantly on how they are used. We recommend using syntactic preprocessors or enforcing disciplined usage. If used sparely in a disciplined fashion, and if possible with suitable tool support, such as the extensions we discuss in Chap. 7, many problems can be avoided.

Summary preprocessors
Strong points:

- Easy to use, well-known.
- Simple programming model: annotate and conditionally remove.
- Compile-time customization of the source code. No boilerplate code (see Sect. 3.1.1, p. 48).
- Flexible and supports arbitrary granularity (see Sect. 3.2.5, p. 59).
- Little preplanning required (see Sect. 3.2.1, p. 53).
- Features usually traceable to (several) code locations (see Sect. 3.2.2, p. 54).
- Lightweight mechanism for extractive product line adoption.



```
    void privTaskInit(Tuint0
rivTrace(traceTaskInit | uiTaskN
TtaskControlBlock * taskTCB = priv
#if (defReUseTaskInit == cfgTrue)
  if ((uiInitControl & defInitLockMa
  {
    #if (cfgUseSynchronization != cf
      if (uiTaskNumber < defNumberOf
      {
        privCleanSlotStack((TtaskExt
        #if ((defUseMutexes == cfgT
          privReleaseSyncBlockingT
        #endif
        }
    ndif
        /cfgUseFileSystem
```

Fig. 5.17 Preprocessor directives in the code of *Femto OS*: highlighted lines represent preprocessor directives such as #ifdef, white lines represent the remaining C code, comment lines are not shown

```
static int __rep_queue_filedone(dbenv, rep, rfp)
  DB_ENV *dbenv;
  REP *rep;
  __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
  COMPQUIET(rep, NULL);
  COMPQUIET(rfp, NULL);
  return __db_no_queue_am(dbenv);
#else
  db_pgno_t first, last;
  u_int32_t flags;
  int empty, ret, t_ret;
#ifdef DIAGNOSTIC
  DB_MSGBUF mb;
#endif
  // over 100 lines of additional code
}
#endif
```

Fig. 5.18 Adapted code excerpt of Berkeley DB, which contains a syntax error in products without Have_Queue

- Uniform application to source code and noncode artifacts (see Sect. 3.2.6, p. 60).

Weak points:

- Scattering and tangling of feature code and configuration knowledge. No clear separation of concerns (see Sect. 3.2.3, p. 55),
- No support for information hiding (see Sect. 3.2.4, p. 57).
- May obfuscate source code.
- Often used in ad hoc or undisciplined fashion.
- Prone to simple errors.
- Difficult to analyze and to write tool support for the underlying language.

5.4 Further Reading

Although version-control systems are typical in practice, there is little literature on using them for variability implementation. A notable exception is the experience report by Staples and Hill (2004), who report on using a combination of branches and build-system scripts to implement variability. Issues regarding comparing and merging revisions and the role of structure are discussed by Mens (2002) and Apel et al. (2011, 2012b).

Build systems are discussed more broadly in the literature. A typical problem is that build systems become complex and slow in many large projects, so there exists quite some work on visualizing and reengineering build systems. Adams et al. (2007)

offer a good introduction to challenges and reengineering of build systems. In the context of product line development, the build system of the Linux kernel is well-documented and was studied intensively in the recent years (Adams et al. 2008a; Berger et al. 2010a; Nadi and Holt 2010; Dietrich et al. 2012a). Especially, Adams et al. (2008a) reflect on the challenges of the build system and illustrate how the Linux developers have repeatedly attempted simpler and more disciplined strategies for their build system. All these pieces of work demonstrate the difficulty of statically analyzing build systems.

There is plenty of literature on preprocessors, mostly criticizing their deficits and recommending alternatives or tools. For example, Spencer and Collyer (1992) and Favre (1995, 1997) wrote critical articles on preprocessor usage; Ernst et al. (2002) have analyzed C source code in the wild and identified many pitfalls especially regarding macros; Liebig et al. (2011) discuss problems in building tool support and how disciplined annotations can help; and Kästner (2010) has collected a detailed catalog of advantages and disadvantages of preprocessors and proposals for corresponding tool support. In Chaps. 7 and 10, we come back to preprocessors and discuss advanced tool support and analysis methods.

Finally, Sommerville (2010, Chap. 25) provides a broader overview of software configuration management, beyond variability implementation. Software configuration management encompasses all techniques discussed in this chapter and many more, such as change management and release planning.

Exercises

5.1 Implement the chat system (Exercise 4.1, page 96) with a combination of a version-control and a build system. For Java, we recommend to try a combination of *Subversion* and *ANT*. Build at least three different products.

(a) Discuss different strategies of using branches for implementing the chat system's variability.
(b) Reflect on code quality and granularity of the resulting implementation.
(c) Modify the chat system, such that (a) the application prints its version to the command line during initialization, (b) encryption uses a different key. Can these changes be made locally?
(d) Does the implementation replicate code? Discuss possible mechanisms to avoid code replication.

5.2 Find an open-source project with a reasonable large history and developer base (for example, homebrew,[5] node,[6] and Stellarium[7]). Investigate how the version-control system is used in the project. Are branches used for patches, for developing

[5] https://github.com/mxcl/homebrew

[6] https://github.com/joyent/node

[7] http://sourceforge.net/projects/stellarium/

features, for variability, or for other reasons? What code and noncode artifacts are tracked? Are branches merged again, are patches applied to multiple branches? Are there any policies around using branches? Do merge conflicts arise frequently and are they hard to resolve?

5.3 Find an open-source project that uses a build script (for example, make, ANT, maven). Possible example projects are Rhino,[8] PHP,[9] and TinyOS.[10] Analyze the tasks performed by the build system. Does the build system manage variability by selecting which files to generate, which files to compile, or how to compile files? Does the build system orchestrate other variability mechanisms? Is there variability in the dependencies of the project? Is there variability in noncode artifacts? What effort is required to trace features in the build process?

5.4 Implement the variability of the chat system (Exercise 4.1, page 96) with preprocessor directives. Aim at minimal code size (code not needed in a product should be removed by the preprocessor). When using Java, we recommend to use *FeatureIDE* with the *Munge* or *Antenna* plug-in (see Appendix A).

(a) Reflect on code quality and implementation effort of the resulting system critically.
(b) Add a new feature Console that replaces the graphical user interface with a minimalistic command-line interface. Reflect on the required code changes (at least regarding invasiveness, effort, and code quality).

5.5 Find an open-source project that uses conditional compilation with a preprocessor (for example, Vim,[11] Berkeley DB,[12] MobileRSSReader,[13] and MobileMedia[14]). Investigate how the processor is used. Select three features and investigate how those are implemented in the source code.

(a) On what code and noncode artifacts is the preprocessor used?
(b) Does conditional compilation control variability in the sense of product line features or does it serve a different purpose?
(c) At what granularity is conditional compilation used?
(d) Is conditional compilation used in a disciplined way? Find three examples of disciplined and undisciplined use (if available).
(e) Is the implementation of features scattered? Is any effort for separation of concerns recognizable?
(f) Discuss the benefits and drawbacks of using the preprocessor as variability mechanism in the context of this project.

[8] https://github.com/mozilla/rhino
[9] https://github.com/php/php-src
[10] http://www.tinyos.net/
[11] http://www.vim.org/
[12] http://www.oracle.com/technetwork/products/berkeleydb/
[13] http://code.google.com/p/mobile-rss-reader/
[14] http://mobilemedia.sourceforge.net/

5.6 Explain the difference between lexical and syntactic preprocessors and provide corresponding examples. What are their mutual strengths and weaknesses? What might be a reason that most preprocessors used in practice are lexical preprocessors?

5.7 Locate and classify the errors in the following three code snippets. Discuss the role of the preprocessor for introducing and locating errors. What is the relation to annotation discipline?

```
int a = 1;
int b = 0;
#ifdef A
int c = a;
#else
char c = a;
#endif
if (c) {
#ifdef B
  c += a;
  c /= b;
}
#endif
```

(a)

```
int a = 1;
int b = 0;
#ifdef A
char c[] = a;
#else
int c = a;
#endif
if (c) {
#ifdef B
  c += a;
  c /= b;
#endif
}
```

(b)

```
int a = 1;
int b = 0;
#ifdef A
int c = a;
#else
char c = a;
#endif
if (c) {
  c += a;
#ifdef B
  c /= b;
#endif
}
```

(c)

5.8 Reconsider the scenarios of Exercise 2.9 (page 43). Which implementation approach would you recommend to the developers and why? Would you give additional advice on how to use these implementation mechanisms?

5.9 Extend the comparison of Exercise 4.11 (page 101) with the additional implementation strategies from this chapter.

Chapter 6
Advanced, Language-Based Variability Mechanisms

After reading the chapter, you should be able to

- explain the key concepts of collaboration-based design and feature-oriented programming,
- understand the key mechanisms of AspectJ and write simple aspects in this language,
- implement product lines with feature-oriented and aspect-oriented languages and their combination,
- discuss trade-offs between these and previous implementation techniques,
- contrast feature-oriented and aspect-oriented languages regarding their key mechanisms,
- select a suitable implementation technique for a given product line,
- critically discuss the conflict between preplanning and obliviousness, and
- discuss strategies of developing feature-oriented extensions of other code and noncode languages.

Motivated by the shortcomings of classic implementation techniques (surveyed in Chaps. 4 and 5) new techniques have been proposed. Their goal is to provide efficient means to implement product lines, to make their variability, features, and interactions explicit, as well as to ease their handling and the reasoning about them.

Again, we distinguish between two types: those that extend a language to express feature-oriented concepts and those that provide tool support to develop and manage features. In this chapter, we examine advanced, language-based approaches. Specifically, we introduce and discuss two in detail: feature-oriented programming and aspect-oriented programming, both of which are composition-based. We discuss their mutual strengths and weaknesses and present an approach to combine them. Finally, we briefly review other notable language-based approaches. Almost all approaches discussed in this and the next chapter support static (that is, compile-time) variability only and are largely composition-based.

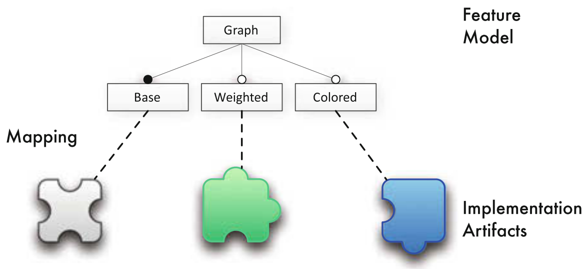


Fig. 6.1 Mapping between features and implementation artifacts in feature-oriented programming

6.1 Feature-Oriented Programming

Feature-oriented programming is a composition-based approach for building software product lines that relies directly on the notion of features. The idea is to decompose a system's design and code into the features it provides (Prehofer 1997; Batory et al. 2004). This way, the structure of a system aligns with its features, ideally, one module or component per feature. To this end, new language constructs are needed to express which parts of a program contribute to which features and to encapsulate the feature's code in composable, modular units.

Feature modularity[1] has many benefits, in particular, a simple mapping between features and their implementations, which eases tracing and debugging (as we will discuss in Sect. 6.1.5). Once a user selects a set of features, the corresponding implementation modules are composed automatically. Figure 6.1 illustrates the (ideally) clean mapping that feature-oriented programming establishes between features and their implementing artifacts.

6.1.1 Collaboration-Based Design

Collaborations lie at the center of feature-oriented design, and whose concept has been known for over twenty years (Reenskaug et al. 1992). To motivate collaborations, we start with a simple, nonprogramming example: the *mentor-student collaboration*.

A person in the role of a mentor has responsibilities to instruct students on certain topics, guide their development, and provide a stimulating environment in which to learn. A person in the role of a student has responsibilities to study the offered material, attend lectures, complete homework on time, and to seek an appreciation for what is being presented, asking questions otherwise. Some persons play multiple

[1] There is an ongoing discussion on the role and goals of modularity (Ostermann et al. 2011); we use a liberal interpretation to which modules are a code structuring mechanism; see Sect. 3.2.4.

roles, for example, student and mentor simultaneously (for example, Niels Bohr simultaneously played the role of student with Ernest Rutherford and the role of mentor to Werner Heisenberg—all famous physicists and Nobel laureates). Much like in the real world, collaborations are ubiquitous in software.

Collaboration-based design is a fundamental technique to decompose systems into collaborations (Reenskaug et al. 1992; VanHilst and Notkin 1996; Smaragdakis and Batory 2002). A collaboration is a set of interacting classes, each class playing a distinct role, to achieve a certain function or capability. For example, to implement a graphical user interface, typically, a large number of classes collaborate, each playing a different role: capture events, represent buttons, action handling, and so on. A collaboration can implement a feature, but not all collaborations implement features.

Feature-oriented programming and collaboration-based design can be applied on top of a wide variety of host languages, including object-oriented, functional, and domain-specific languages (Apel et al. 2013a). We use Java in our examples, but the underlying concepts are largely language independent.

Definition 6.1 A *collaboration* is a set of classes (or parts thereof) that cooperate to implement the functionality of a feature. □

Typically, a software system consists of multiple collaborations implementing multiple features. So, a class often participates in the implementation of multiple features. That is, a class plays multiple roles in multiple collaborations with other classes.

Definition 6.2 A *role* defines the responsibilities a class takes in a collaboration. □

A class that plays multiple roles defines multiple sets of functionalities (sets of methods and fields) associated with the individual roles. Separating the different roles of a class as well as bundling all roles that belong to a collaboration are key objectives of collaboration-based design and feature-oriented programming.

Example 6.1 In Fig. 6.2, we show a sample collaboration-based design inspired by the graph library. The diagram uses UML-like notation with some extensions: rows (grey boxes) denote collaborations; white boxes represent classes or roles; solid arrows (that link classes column-wise) denote the application of a new role to a class.

Let us interpret Fig. 6.2 row-wise: Collaboration BasicGraph consists of the classes Graph, Node, and Edge, which together provide the functionality to construct and display graph structures. Collaboration Weighted adds roles to the classes Graph and Edge as well as a new class Weight; Weighted extends the graph implementation to support weighted edges. Collaboration Colored adds roles to the classes Node and

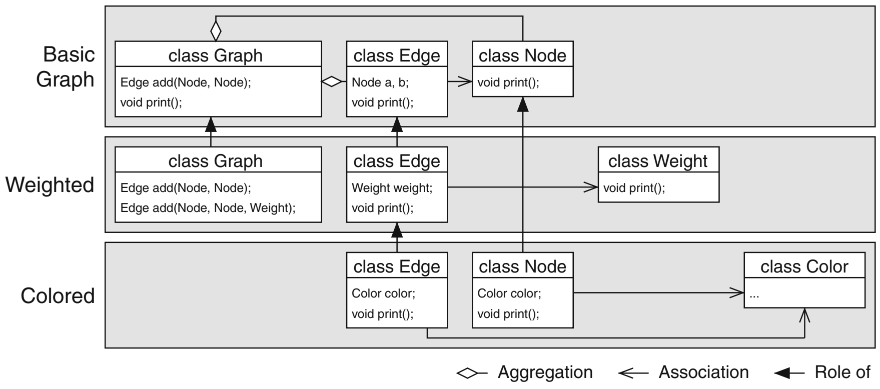


Fig. 6.2 Collaboration-based design of a simple graph implementation

Edge as well as a new class Color; it provides support for colored output of nodes and edges on the command line.

Now a column-wise interpretation: The Graph class simultaneously plays two roles in two different collaborations (BasicGraph and Weighted). Class Edge simultaneously plays three different roles in all three collaborations. Class Node simultaneously plays roles in collaborations BasicGraph and Weighted, and class Weight and class Color play their part in individual collaborations.

Composing collaborations is, in effect, superimposing them by lining-up classes according to the roles they play. □

6.1.2 Feature Modules

In feature-oriented programming, each collaboration maps to a feature and is called a *feature module*. Different combinations of feature modules satisfy different needs of customers or application scenarios. For example, based on the design shown in Fig. 6.2, we can create four different graph products by composing different sets of collaborations, a basic graph, a weighted graph, a colored graph, and a weighted and colored graph.

In Fig. 6.2, we also illustrate how features crosscut a given (object-oriented) program structure. A feature module refines the content of a base program (which itself may result from a composition of feature modules) either by adding new elements or by modifying and extending existing elements. Hence, the order in which features are applied is important: earlier features in the sequence may add elements that are refined by later features. This is typical for the step-wise development of programs.

```
layer BasicGraph;

class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  void print() {
    for(int i = 0; i < edges.size(); i++) {
      ((Edge)edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}
```

```
layer BasicGraph;

class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() {
    System.out.print(id);
  }
}
```

```
layer BasicGraph;

class Edge {
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
  }
}
```

Fig. 6.3 A simple graph implementation (feature module BasicGraph)

6.1.3 The Jak Language

Jak is an extension of Java for feature-oriented programming (Batory et al. 2004). In Fig. 6.3, we show the Jak implementation of our graph example as a feature module. It consists of the classes Graph, Node, and Edge. Except keyword layer, which denotes the feature a class belongs to, the Jak code in Fig. 6.3 is not different from plain Java code.

Roles that extend existing classes are implemented using class refinements, denoted by keyword refines. A *class refinement* can add new members to a class

```
layer Weighted;

refines class Graph {
  Edge add(Node n, Node m) {
    Edge e = Super.add(n, m);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
  }
}
```

```
layer Weighted;

refines class Edge {
  Weight weight;
  void print() {
    Super.print();
    weight.print();
  }
}
```

```
layer Weighted;

class Weight {
  void print() { /* ... */ }
}
```

Fig. 6.4 Extending the basic graph implementation by introducing weights to edges (feature module Weighted)

and extend existing methods. A method extension is implemented by method overriding and calling the overridden method via the keyword Super.[2]

In Fig. 6.4, we show feature Weighted implemented in Jak. It introduces a new class Weight that represents the weight of an edge (bottom) and refines (applies a role to) class Graph (top) by introducing a new method add that assigns a given weight value to an edge, and by overriding the existing method add to assign a default weight value. Furthermore, it refines class Edge (middle) by adding a field and by extending the print method to display the weight.

Class refinement is a form of *mixin-based inheritance* (Bracha and Cook 1990; Flatt et al. 1998), in which subclasses, called mixins, are abstract in the sense that they can be applied to different concrete superclasses (which is not possible with subclassing in Java). In the graph example, the mechanism of class refinement gives us the flexibility to refine either class Edge of feature BasicGraph or its refinement applied by Weighted. Mixins are the static counterpart to the Decorator design pattern, discussed in Sect. 4.2.4. They overcome the problems of inheritance with regard to step-wise

[2] Jak's keyword Super is similar to Java's keyword super. While Super refers to the method that has been overridden by a class refinement, super refers to the method that has been overridden by a subclass. To avoid confusion, other feature-oriented languages use instead keywords such as original (Apel et al. 2009).

development and code reuse (see Fig. 4.9). Mixin layers have been introduced to bundle multiple mixins that form a semantically coherent unit (Smaragdakis and Batory 2002). Feature modules are rooted in the concept of mixin layers. A feature module bundles all classes and class refinements that contribute to the implementation of a feature.

The semantics of feature-module (and mixin-layer) composition is as follows: (1) all classes of all feature modules are assembled in a single program; (2) each class is merged with all of its refinements in that all members of the refinements are added to the class; (3) in the case an existing method is overridden (that is, a refinement contains a method that is already present), the overridden method can be called by the overriding method via keyword Super. The actual composition mechanism at the technical level may vary (Batory et al. 2004). For example, refinements may be implemented by subclassing with name mangling or by mixin composition (Smaragdakis and Batory 2002). Furthermore, different implementations handle special cases such as field overriding differently.

In Jak and in other contemporary feature-oriented programming languages and tools, feature modules are represented by file-system directories, called *containment hierarchies*; classes and their refinements are stored in files inside the corresponding containment hierarchies. Programmers select features by name before compilation via command-line parameters or tool support. In Fig. 6.6, we show a snapshot of the containment hierarchies and the feature model of the graph example in the development environment *FeatureIDE* (see Appendix A). For each feature, there is a directory that contains the files with the classes and refinements implementing the features.

6.1.4 Models of Feature-Oriented Programming

To abstract from implementation details, researchers have developed models of feature-oriented programming that capture essential properties and emphasize its generality.

GenVoca is an algebraic model for feature-oriented programming (Batory and O'Malley 1992). Each product line is modeled by a corresponding algebra, which is called its *GenVoca model*. For example, 'Graph = {BasicGraph, Weighted, Colored}' denotes a model Graph that has the features BasicGraph, Weighted, and Colored.

Features are modeled as program transformations (a.k.a. functions that map programs to programs). A *constant function* (a.k.a. *constant*) represents a base program. All other functions are unary (single-parametered) that receive a program as input and return a modified program as output. The returned program is the original plus

```
class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
  }
  void print() {
    for(int i = 0; i < edges.size(); i++) {
      ((Edge)edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}
```

```
class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() {
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
    weight.print();
  }
}
```

```
class Weight {
  void print() { /* ... */ }
}
```

Fig. 6.5 Composing the feature modules BasicGraph and Weighted

the changes that are needed to add the designated feature. That is, functions represent program refinements that implement features. For example, ‘Weighted • X’ adds feature Weighted to program X, where ‘•’ is function composition. Similarly, ‘Colored • X’ adds Colored to X. The design of a software product is a named *feature expression*, for example:

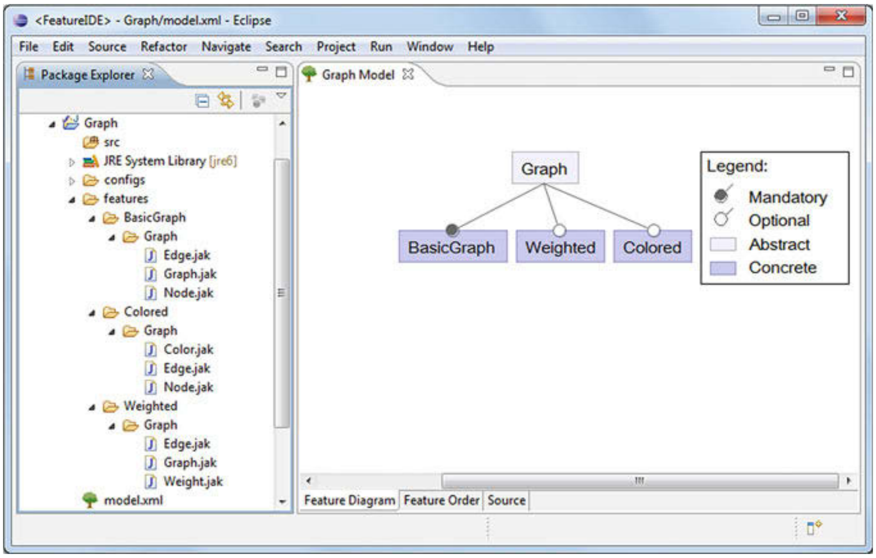


Fig. 6.6 Containment hierarchy (left) and feature model (right) of the graph example

$$\textsf{WeightedGraph} = \textsf{Weighted} \bullet \textsf{BasicGraph}$$
$$\textsf{ColoredWeightedGraph} = \textsf{Colored} \bullet \textsf{Weighted} \bullet \textsf{BasicGraph}$$

AHEAD (Algebraic Hierarchical Equations for Application Design) is the successor of *GenVoca* (Batory et al. 2004). It scales the ideas of GenVoca to all kinds of software artifacts, thus applying the principle of uniformity (see Sect. 3.2.6) to feature-oriented programming. That is, a feature consists not only of source code but of all artifacts that contribute to that feature, including documentation, test cases, design documents, makefiles, performance profiles, and mathematical models.

The AHEAD tool suite implements these ideas. It contains several tools for developing, debugging, and composing code and noncode artifacts. As said previously, each feature is represented by a containment hierarchy, which is a directory that maintains a substructure organizing the feature's artifacts (see Fig. 6.6). Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts recursively by name and type (see Fig. 6.8 for an example). For each artifact type, a different implementation of the *composition operator* '•' (that is, a tool that performs the composition) has to be provided in AHEAD, much like Jak for Java artifacts. For example, the graph implementation of Fig. 6.2 may be paired with documentation in HTML, as illustrated in Fig. 6.7.

Distinct composition tools have been created to work with particular kinds of software artifacts. That is, there is a special tool for defining and composing Jak representations of programs, there is another special tool for defining and composing XML artifacts, and so on. For each artifact type, at least one special tool has to be built.

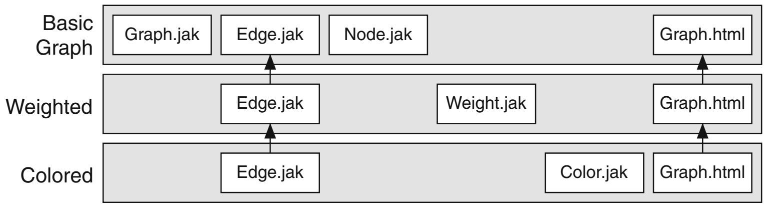


Fig. 6.7 Collaboration-based design of the graph library including HTML documentation

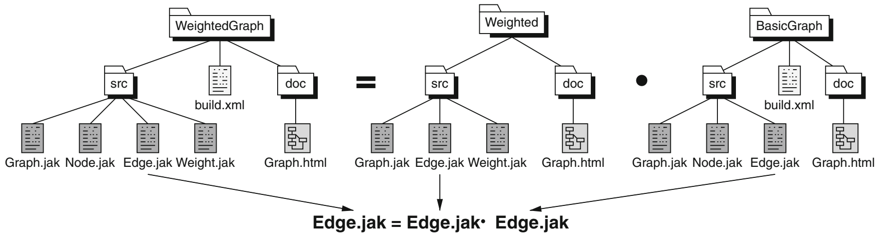


Fig. 6.8 Composing containment hierarchies of the graph library (Apel 2007)

The AHEAD tool suite brings these separate tools together and selects different tools for different kinds of files during feature composition, establishing a clear interface to the build system. Composing Jak files will invoke a Jak-composition tool, whereas composing XML files invokes an XML-composition tool, and so on, as illustrated in Fig. 6.8.

Following the philosophy of *AHEAD*, the *FeatureHouse* tool suite has been developed that allows programmers to enhance given languages rapidly with support for feature-oriented programming, for example, C#, C, JavaCC, Haskell, Alloy, and UML (Apel et al. 2009). *FeatureHouse* is a framework for software composition supported by a corresponding tool chain. It provides facilities for feature composition based on a language-independent model and tool chain for software artifacts, and a plug-in mechanism for the integration of new artifact languages. A language plug-in is essentially the language's grammar plus some further information on how different structural elements are composed. The benefit of this generality is that it is now substantially easier to build new languages and tools for feature-based development.

6.1.5 Discussion

Like all approaches in this chapter, feature-oriented programming is a language-based and composition-based approach to product-line implementation (see *language-based versus tool-based* and *annotation versus composition* in Sects. 3.1.2 and 3.1.3,

p. 49 and 50). The feature modules can then be composed statically at compile time or even dynamically at run time (Rosenmüller et al. 2011) (see *binding times* in Sect. 3.1.1, p. 48). Although some static composition mechanisms encode class refinement as inheritance, there are composition mechanisms that entirely inline extensions into the base code without run-time overhead.

Feature-oriented programming languages provide a means to collect the entire implementation of a feature cohesively in a feature module. Every piece of code that contributes to the implementation of a feature is included in the corresponding feature module, which can be referred to by the feature's name (see *separation of concerns* in Sect. 3.2.3, p. 55).

Feature-oriented programming excels at feature traceability. There is a direct mapping from features to feature modules that implement them (see *feature traceability* in Sect. 3.2.2, p. 54). Attaining a similarly simple traceability is difficult with classic implementation approaches. For example, using run-time parameters and preprocessor directives, a feature's code is typically scattered all across the code base. Frameworks and components alleviate this problem by leveraging encapsulation and extension mechanisms such as interfaces and inheritance, but the lack of crosscutting modularity makes it difficult to really separate a feature's code from the code of other features (see Chap. 4).

In feature-oriented programming, a feature is implemented by a collaboration of classes contained in a feature module. This way, feature-oriented programming supports the separation of crosscutting concerns, which eases feature tracing: What would be otherwise scattered, is encapsulated in a single place. As we will see in Sect. 6.3, there are further kinds of crosscutting concerns that are not so well supported by feature-oriented programming.

In Chap. 4, we have learned that, using classic implementation approaches, many additions and modifications a feature has to apply, have to be planned upfront when designing the extension points (see *preplanning* in Sect. 3.2.1, p. 53.). That is, a programmer has to anticipate at which places a feature will "hook into" the system, or she has to manually alter the existing code.

Using feature-oriented programming, one can extend existing classes via class refinement, without touching existing implementations. In contrast to inheritance, one can refine a class without breaking any client code. For example, in Jak, we can add a field weight to class Edge transparently: Every reference to an Edge object points immediately to the extended version. Using inheritance (for example, in Java), we can also add fields, but then we have to introduce a new type (for example, class WeightedEdge), and every instantiation of Edge objects have to be replaced with the new type, as illustrated in Fig. 6.9. This can be controlled by the template-method pattern, but only if that extension was anticipated by the programmer.

Once we have multiple optional features, we want to compose them in different combinations, for example, a simple graph, a weighted graph, or a weighted and colored graph. Using classic approaches that rely on inheritance and delegation, this is difficult, as discussed in Fig. 4.9, p. 78. As we have seen there, this leads to exploding class hierarchies, or induces other problems such as the diamond problem

```
class Edge { /* ... */ }

/* ... */

// extensions of Edge take no effect
void foo() {
  Edge edge = new Edge();
}

/* ... */
```

```
// extending class Edge
class WeightedEdge extends Edge {
  Weight weight;
}
```

```
layer BasicGraph;

class Edge { /* ... */ }

/* ... */

// extensions of Edge take effect
void foo() {
  Edge edge = new Edge();
}

/* ... */
```

```
layer Weighted;
// extending class Edge
refines class Edge {
  Weight weight;
}
```

Fig. 6.9 Inheritance versus class refinement

(Smaragdakis and Batory 2002). With class refinement, one can flexibly combine extensions.

Although specific tools and language extensions for feature-oriented programming extend specific programming languages, such as Jak for Java, the underlying concepts are more general, as we have discussed in Sect. 6.1.4. Hence, feature-oriented programming is possible in most languages, which allows refining code and noncode artifacts uniformly (see *uniformity* in Sect. 3.2.6, p. 60).

Most languages for feature-oriented programming, including Jak, compose code at the level of methods (see *granularity* in Sect. 3.2.5, p. 59), which is finer than the granularity of most traditional language-based approaches. Refinements can introduce new methods and override existing methods. Injecting changes at a finer level of granularity is not possible though. In different languages, different levels of granularity are possible, but granularity is limited by the need to find structures with addressable names for composition (Apel et al. 2013a).

Finally, it is important to note that feature-oriented programming is no silver bullet. Its composition model is syntax-directed, which is the price for the degree of uniformity and generality embodied in feature-oriented tools and languages. There is no support, yet, for information hiding in the sense that each feature module has an explicit contractual interface and protects internal implementation details (see *information hiding* in Sect. 3.2.4, p. 57). Though recent proposals for byte-code based composition and access control at the feature level aim at closing this gap (Apel et al. 2012a). With regard to crosscutting, feature-oriented programming is superior to object-oriented programming, but falls behind the capabilities of aspect-oriented programming, as we will discuss in Sect. 6.3. Finally, the ideal goal of mapping each feature to a single feature module is hard to achieve in practice, especially, when features depend structurally on each other and interact. We discuss this issue in Chap. 9.

Summary feature-oriented programming
Strong points:

- Easy to use language-based mechanism, requires only minimal language extensions.
- Compile-time customization of source code without run-time overhead (see Sect. 3.1.1, p. 48).
- Separation of (possibly crosscutting) feature code into distinct feature modules (see Sect. 3.2.3, p. 55).
- Direct feature traceability from a feature to its implementation in a feature module (see Sect. 3.2.2, p. 54).
- Conceptually uniformly applicable to code and noncode artifacts, tools already cover many languages (see Sect. 3.2.6, p. 60).
- Little preplanning required due to mixin-based extension mechanism (see Sect. 3.2.1, p. 53).

Weak points:

- Requires adoption of a language extension or unfamilar composition tools as part of the development process.
- Granularity at the level of methods (or other named structural entities) (see Sect. 3.2.5, p. 59).
- Composition is syntax-directed and does not offer enforced interfaces between feature modules (see Sect. 3.2.4, p. 57).
- Tools need to be constructed for every language (see Sect. 3.2.6, p. 60), but these may be generated (see Sect. 6.1.4, p. 135).
- Only academic tools so far, little experience in practice.

6.2 Aspect-Oriented Programming

Aspect-oriented programming aims at the modularization of crosscutting concerns (Kiczales et al. 1997). In our discussion of feature-oriented programming, we have already considered a kind of crosscutting concern: A collaboration extends a program at different places, thus it cuts across the module boundaries introduced by classes. Feature modules implement collaborations in a cohesive way. While feature-oriented programming has been developed as a feature-implementation (collaboration-implementation) technique, aspect-oriented programming targets crosscutting concerns from a different perspective, as we will explain shortly.

As we have seen in earlier chapters, crosscutting leads to code scattering and tangling. In Fig. 6.10, we show how the implementation of feature Colored is scattered across the basic graph implementation. The implementation of feature Colored is scattered across three classes, and inside these classes it affects two methods (code associated with feature Colored is highlighted). Notice also a certain amount of code

```
class Graph { /* ... */ }
class Node {
  Color color = new Color();
  Color getColor() { return color; }
  int id = 0;
  Node(int _id) { id = _id; }
  void print() {
    Color.setDisplayColor(getColor());
    System.out.print(id);
  }
}
class Edge {
  Color color = new Color();
  Color getColor() { return color; }
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    Color.setDisplayColor(getColor());
    System.out.print(" (");
    a.print();
    System.out.print(", ");
    b.print();
    System.out.print(") ");
  }
}
class Color {
  static void setDisplayColor(Color c) { /* ... */ }
}
```

Fig. 6.10 Scattered and replicated code of feature Colored (highlighted)

replication: code for managing and changing colors is replicated in the classes Edge and Node. Aspect-oriented programming aims at reducing such code scattering, tangling, and replication induced by concerns that are not well-separated.

6.2.1 Aspects: Separating Crosscutting Concerns

Aspect-oriented programming addresses the problems caused by crosscutting concerns as follows: concerns that can be localized well using classic implementation mechanisms, such as classes and methods, are implemented further using these mechanisms; all other concerns that crosscut the implementation of other concerns are implemented as *aspects*.

An aspect is a programming construct that encapsulates the implementation of a crosscutting concern. It enables code that is associated with one crosscutting concern to be localized into one code unit, thereby eliminating code scattering and tangling. Moreover, aspects can affect multiple other concerns with one piece of code, thereby avoiding code replication.

Definition 6.3 An *aspect* encapsulates the implementation of a crosscutting concern. □

The base program and the aspects are combined using an aspect weaver, forming an executable program.

Definition 6.4 An *aspect weaver* merges the separate aspects of a program and the remaining program elements at user-selected program locations, called *join points*. This process is called *aspect weaving*. □

In Fig. 6.11, we illustrate the weaving of two aspects into a base program consisting of three components. The result is a program that contains the functionality of all components and all aspects, including the effects the aspects have on the structure and behavior of the components. In some sense, the aspect weaver produces again tangled and scattered code, but hidden from the programmer.

Definition 6.5 A *join point* is an event in the execution of a program at which aspects can be woven into the program. The source code locations that give rise to a join point are called its *join-point shadows*. □

Join points are, for instance, the execution or call of a method, the access of a field, the raise of an exception, and the initialization of a class. Join point shadows are, for example, statements that access fields or invoke methods as well as method declarations.

In most aspect-oriented languages, aspects contain *pointcuts*, *advice*, and *inter-type declarations*, or similar language constructs. A pointcut defines which join points an aspect affects. One can think of a pointcut as a filter that selects those join points of all possible join points in a program that are of interest to a certain aspect.

Definition 6.6 A *pointcut* is a declarative specification of the join points that an aspect affects. It is a predicate that determines whether a given join point matches. □

Advice and inter-type declarations are mechanisms to inject and execute code that belongs to the aspect.

Definition 6.7 A piece of *advice* is a method-like element of an aspect that encapsulates the instructions that are executed at a set of join points. Pieces

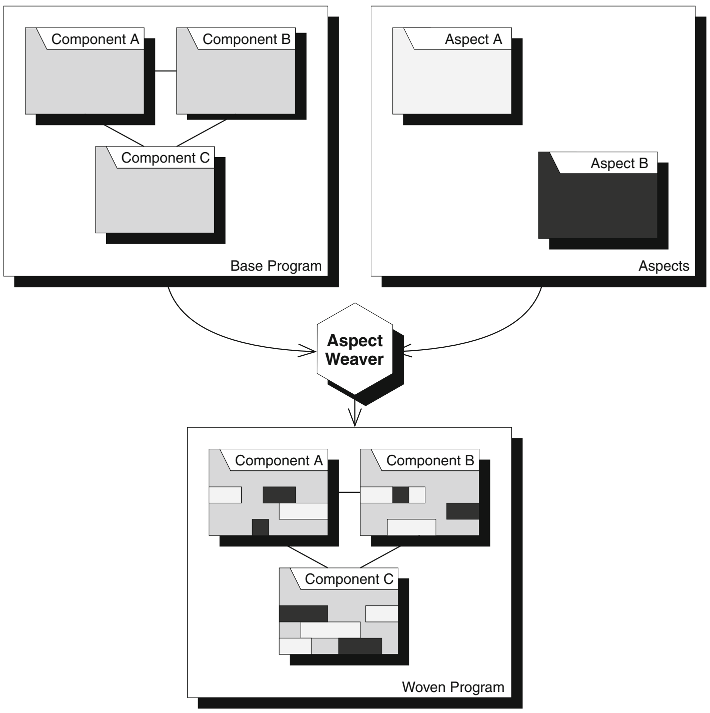


Fig. 6.11 An aspect weaver weaves the aspects into the base program (adapted from Spinczyk 2002)

of advice are bound to pointcuts that define the set of predefined join points being *advised*. □

Definition 6.8 An *inter-type declaration* injects a method, field, or interface from inside an aspect into an existing class or interface. □

Compared to other extension mechanisms, such as inheritance, aspects can *quantify* over whole sets of join points (Filman and Friedman 2005). For example, an aspect can advise all method executions of a program or all field accesses from within in a certain package that match a certain type and name pattern. From a weaving perspective, the code associated with the aspect is woven into many locations of the base program.

Definition 6.9 *Quantification* is the process of selecting multiple join points based on a declarative specification (that is, based on a pointcut). □

6.2.2 The AspectJ Language

AspectJ[3] is an aspect-oriented language extension of Java. It is the most popular and widely used aspect-oriented language. Many of the aspect-oriented concepts discussed so far have been introduced first by AspectJ.

In Fig. 6.12, we illustrate how an aspect in concert with a class and an interface implements our feature Colored. Dashed arrows point to the structural elements of the graph implementation affected by the aspect (only a subset is depicted). The AspectJ weaver weaves the aspect into the basic graph implementation, thus creating a graph implementation incorporating colors.

In Fig. 6.13, we depict one possible implementation of feature Colored in AspectJ. Aspect Colored defines an interface IColored for all classes that maintain a color value (Line 2) and declares via inter-type declaration that the classes Node and Edge implement that interface (Line 3) using the keywords declare parents. Furthermore, it introduces a field color and a method getColor by means of two inter-type declarations, which are like Java field or method declarations except that the name of the target type precedes the name of the field or method. Finally, the aspect advises the execution of method print of all colored entities (that is, Edge and Node) to change the display color at run time. Keyword before states that the advice is executed before every selected join point. The pointcut execution(void print()) && this(c) selects all executions of print methods and passes an instance c of a colored entity to the advice body (using this).

In Fig. 6.13, advice is executed before (keyword before) the execution of method print. Alternatively, using the keywords after and around, advice can be executed after

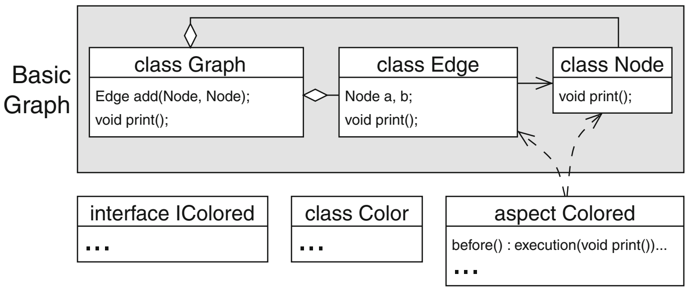


Fig. 6.12 Implementing feature Colored with aspect-oriented programming

[3] http://www.eclipse.org/aspectj/

```
aspect Colored {
  interface IColored { Color getColor(); }
  declare parents: (Node || Edge) implements IColored;
  Color IColored.color;
  Color IColored.getColor() { return color; }
  before(IColored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.getColor());
  }
}
```

Fig. 6.13 Implementing feature `Colored` using *AspectJ*

```
aspect AdviceExample {
  before() : execution(void print()) {
    System.out.println("before print");
  }
  around() : execution(void print()) {
    System.out.println("around print (before)");
    proceed();
    System.out.println("around print (after)");
  }
  after() : execution(void print()) {
    System.out.println("after print");
  }
}
```

Fig. 6.14 Examples of before, around, and after advice

or instead of the advised join point. In Fig. 6.14, we give examples for all three kinds of advice.

The output of the program of Fig. 6.14 illustrates the execution order of different kinds of advice:

```
before print
around print (before)
... // output of method print
around print (after)
after print
```

In the case of around advice, keyword proceed is used to trigger the advised method execution (or other kind of join point).

Beside three types of advice, there are many predefined pointcuts that can be used and logically combined to select different types of join points, for example, call for method calls, set and get for field accesses, cflow for control-flow related join points, and so on. AspectJ is a powerful language that offers many more mechanisms and syntaxes to express them. Rather than explaining them, which is done elsewhere to a large extent (Laddad 2003), we illustrate the power by means of examples. In Fig. 6.15, we show and explain typical patterns to capture entire sets of join points, and in Fig. 6.16, we show and explain four typical aspects.

```
aspect SomePointcutPatterns {
  // select specific method: int twice(int) in class MathUtil
  pointcut p0() : execution(int MathUtil.twice(int));
  // ...any return type
  pointcut p1() : execution(* MathUtil.twice(int));
  // ...any parameter type
  pointcut p2() : execution(int MathUtil.twice(*));
  // ...any combination of parameters
  pointcut p3() : execution(int MathUtil.twice(..));
  // ...any method within class MathUtil
  pointcut p4() : execution(int MathUtil.*(int));
  // ...any method whose name starts with 'tw'
  pointcut p5() : execution(int MathUtil.tw*(int));
  // ...any method twice in any class
  pointcut p6() : execution(int *.twice(int));
  // ...any method twice in any subclass of MathUtil
  pointcut p7() : execution(int MathUtil+.twice(int));
  // ...any method twice in package com
  pointcut p8() : execution(* com..twice(..));
  // ...any method of a program
  pointcut p9() : execution(* *..*(..));
}
```

Fig. 6.15 Typical patterns used in AspectJ pointcuts

6.2.3 Aspects for Product Lines

Aspect-oriented programming can be used to develop feature-oriented product lines. The straightforward approach is to implement one aspect per feature. Based on a user's feature selection, the corresponding aspects are included in the weaving process, possibly controlled by a build system.

Developers using the above pattern inevitably encounter the following problem: Aspects encapsulate changes that need to be made to *existing* classes. Aspects do *not* encapsulate new classes and packages that a feature introduces; they allow for the introduction of nested classes within an aspect, but this is often unsatisfactory (and no solution for package introductions). Implementing a feature using multiple aspect files (and, possibly, class and package introductions), the weaving process becomes more complicated, because it has to keep track of which aspects and parts thereof contribute to the selected features. So, developers often invent their own form of feature modularization to encapsulate the set of aspects, new classes and packages, new artifact files and directories, and so on that define the changes that a feature makes to a program (Hunleth and Cytron 2002; Apel et al. 2008b).

Example 6.2 Variations in the graph library can be implemented using aspects. The idea is to separate all code specific to features such as Colored and Weighted, so that we have a plain basic graph implementation. Note that classes are introduced by graph features, but these must be nested classes without substantially altering the basic design of the graph example. (In more complex software product lines, the use of nested classes may become too complicated, as references and qualified names have to be adapted consistently.) In Fig. 6.17, we show an aspect-oriented implementation of an excerpt of the graph library. □

```
aspect Profiler {
  /* print execution time for every public method */
  Object around() : execution(public * com.company..*.* (..)) {
    long start = System.currentTimeMillis();
    try { return proceed();
    } finally {
      long end = System.currentTimeMillis();
      printDuration(end - start, thisJoinPoint.getSignature());
    }
  }
}
```

```
aspect ConnectionPooling {
  /* for a connection request, reuse open Connection objects  */
  Connection around() : call(Connection.new()) {
    if (!connectionPool.isEmpty()) return connectionPool.remove(0);
    else return proceed();
  }
  /* put Connections that are not needed anymore in the pool */
  void around(Connection conn) : call(void Connection.close()) && target(conn) {
    connectionPool.put(conn);
  }
}
```

```
abstract class Shape {
  abstract void moveBy(int x, int y);
}
class Point extends Shape { ... }
class Line extends Shape {
  Point start, end;
  void moveBy(int x, int y) { start.moveBy(x,y); end.moveBy(x,y); }
}
aspect DisplayUpdate {
  /* select all operations that change a shape's position */
  pointcut shapeChanged() : execution(void Shape+.moveBy(..));
  /* update display only if there is not already an update triggered */
  after() : shapeChanged() && !cflowbelow(shapeChanged()) { Display.update(); }
}
```

```
aspect Autosave {
  int count = 0;
  /* count the number of executed commands */
  after(): call(* Command+.execute(..)) { count++; }
  /* reset command counter after saving */
  after(): call(* Application.save()) || call(* Application.autosave()) { count = 0; }
  /* invoke autosave after every fifth command execution */
  before(): call (* Command+.execute(..)) {
    if (count > 4) Application.autosave();
  }
}
```

Fig. 6.16 Four typical aspects, based on Laddad (2003): (1) a profiler that measures the execution time of every public method of a package, (2) pooling and reusing open connections, (3) connecting observers and subjects in a shape library, and (4) ensuring that after five commands an autosave operation is performed

Example 6.3 Similar to the graph example, we can implement a product line for data management systems using aspect-oriented programming, starting from a base program, say, written in C or Java, and applying aspects implementing individual features. As a case study, Kästner et al. have refactored the Java version of Berkeley DB (Oracle's embedded database system) into a basic Java version and 149 AspectJ

```
class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) { /* ... */ }
  void print() { /* ... */ }
}

class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() { /* ... */ }
}

class Edge {
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() { /* ... */ }
}
```

```
aspect Weighted {
  Edge Graph.add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
  }
  Weight Edge.weight;
  after(Edge e) : this(e) && execution(void Edge.print()) {
    e.weight.print();
  }
  static class Weight {
    void print() { /* ... */ }
  }
}
```

```
aspect Colored {
  interface IColored { Color getColor(); }
  declare parents: (Node || Edge) implements IColored;
  Color IColored.color;
  Color IColored.getColor() { return color; }
  before(IColored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.getColor());
  }
  static class Color {
    static void setDisplayColor(Color c) { /* ... */ }
  }
}
```

Fig. 6.17 An implementation of the graph library with AspectJ (excerpt); the features Weighted and Colored are implemented as aspects

aspects implementing 38 features (Kästner et al. 2007). Remarkably, they were unable to implement one aspect per feature, because the aspects would have become too large—an issue that we discuss in Sect. 6.3. □

6.2.4 Discussion

Aspect-oriented programming is a language-based and composition-based approach to product-line implementation (see *language-based versus tool-based* in Sect. 3.1.2,

p. 49 and see *annotation versus composition* in Sect. 3.1.3, p. 50).[4] Selected aspects and classes are woven to form the desired product. Different weaving technologies support different binding times (See Sect. 3.1.3, p. 50) including compile-time binding (such in AspectJ) and load-time binding (Sato et al. 2003; Popovici et al. 2003).

The granularity of extensions is driven by the join-point model of the aspect-oriented language. In AspectJ, developers can make flexible extensions at from introducing classes and methods, to extending the behavior of methods calls inside a function (see *granularity* in Sect. 3.2.5, p. 59). However, especially local variables and control structures are not accessible from AspectJ's join point model, or can only be extended with workarounds or invasive preparations (Laddad 2003; Kästner et al. 2007). Overall, AspectJ supports more fine-grained extensions than any other composition-based approach presented in this book, but it does not allow the very fine-grained extensions that some annotation-based approaches support.

Proper feature traceability is achieved by using one aspect to implement one feature (see *feature traceability* in Sect. 3.2.2, p. 54). This way, all code implementing a feature is encapsulated in a single, addressable code location, which is difficult or impossible with classic implementation approaches. In Sect. 6.3, we discuss limitations of this approach compared to using feature modules.

More than feature-oriented programming, aspect-oriented programming is capable of extending a given program noninvasively without the need of planning the extension in advance (see *preplanning* in Sect. 3.2.1, p. 53). Much like class refinement, inter-type declarations can be used to extend existing classes by adding new members, and execution pointcuts can be used to refine existing methods via overriding (using around advice). In addition, aspects can be used to extend a program by intercepting its control flow at arbitrary join points and by executing additional code (advice). For example, an aspect may trigger the execution of advice code when certain fields of the base program are accessed, methods are called, or exceptions are thrown, and so on. This is difficult with feature-oriented programming. In Sect. 6.3, we will discuss how the advice mechanism improves over the simple mechanism of extending existing methods via overriding, as used in feature-oriented programming.

Much like in feature-oriented programming, aspects can be applied in different combinations to a given program. However, due to the more powerful extension mechanism, this process is error prone when the number of aspects increases (Lopez-Herrejon 2006). Although there are attempts to extract code ideas of aspect-oriented programming into language-independent models (Lafferty and Cahill 2003; Mehner and Rashid 2003) and tools (Boxleitner et al. 2009), there is no unifying theory like AHEAD and no aspect-oriented tool that is truly language independent and similarly expressive than AspectJ.

The most controversial concept of aspect-oriented programming is that the base program is *oblivious* with regard to the aspects that "hook into" the system (Filman and Friedman 2005). Although often referred to as modularization mechanism, most aspect-oriented languages violate the principle of information hiding (Lieberherr

[4] Note that AspectJ advice may bind to Java annotations, which makes it effectively a combination of an annotation-based and a composition-based approach.

et al. 2003; Aldrich 2005; Sullivan et al. 2005; Dantas and Walker 2006; Steimann 2006): an aspect may affect internals of other modules directly, possibly even breaking module interfaces.

The idea behind obliviousness is that the developers of the base program implement their concerns as if there were no aspects, and aspect programmers extend then the base program. For example, a business application is developed and then features such as persistence and authentication are added in the form of aspects. The business application does not need to be prepared for the features to add, but also cannot hide its internals from these aspects. In this sense, the obliviousness principle takes the goal of reducing preplanning to an extreme by giving up guarantees of information hiding (see *preplanning* and *information hiding* in Sects. 3.2.1 and 3.2.4, p. 53 and p. 57).

A problem is that if programmers are not aware of extensions, they are also not aware of the possible problems induced by these extensions. Extensions are not visible locally by inspecting just the module at hand. There are no guaranteed invariants of interfaces and no hiding barriers of local information. What if the programmer modifies the base program such that the set of join points changes in undesired and inadvertent ways? This includes situations in which join points are removed accidentally (for example, by renaming a method that is to be advised) and in which join points are captured by aspects accidentally. This problem is also called the *fragile-pointcut problem* (Störzer and Koppen 2004). Suppose we advise the drawing primitives in a chess program (for example, `drawKing`, `drawQueen`, and `drawKnight`) to optimize display update: `after() : execution(void draw*())`. Adding a method `draw` somewhere else to the program (to finish the game without winner), would inadvertently let the display optimization advice advise this method, possibly without being noticed by the programmer. The fragile-pointcut problem is especially daunting as the changes may occur globally, somewhere in the program. It is caused by the fact that quantification (Sect. 6.2.1) is based on syntactic comparisons (for example, the names and types of methods). Obliviousness worsens the fragile-pointcut problem. Because the base programmer does not know about aspects, it is more likely that changes may break aspect bindings and that nobody notices that.

There are several experimental aspect-oriented languages that give up on the notion of obliviousness and introduce interfaces between base code and aspects, thus reestablishing information hiding but also reintroducing preplanning effort (Aldrich 2005; Steimann et al. 2010; Sullivan et al. 2005).

Summary aspect-oriented programming
Strong points:

- Compile-time variability without run-time overhead; load-time variability possible as well (see Sect. 3.1.1, p. 48).
- Separation of (possibly crosscutting) feature code into distinct aspects (see Sect. 3.2.3, p. 55).

- Direct feature traceability from feature to its implementation in an aspect (see Sect. 3.2.2, p. 54).
- Fine granularity based on events during the program execution, depending on the join point model (see Sect. 3.2.6, p. 60).
- Little or no preplanning effort required (see Sect. 3.2.1, p. 53).

Weak points:

- Requires adoption of a sophisticated language extension, including a novel programming paradigm.
- Different aspect-oriented extensions for different code and noncode languages, only experimental uniform models so far (see Sect. 3.2.6, p. 60).
- Composition is syntax-directed and does not offer enforced interfaces between feature modules (see Sect. 3.2.4, p. 57).
- Though conceptually uniform, tools need to be constructed for every language (see Sect. 3.2.6, p. 60).
- Only academic tools so far, little application in practice.

6.3 Aspects and Feature Modules in Concert

We hope that previous sections made clear that aspects and feature modules are powerful mechanisms, but they have different strengths and weaknesses, not to mention design philosophies. We now examine their mutual strengths and weaknesses more systematically to provide programming guidelines about when to use which mechanism and to motivate a combination of aspect and feature modules that enables programmers to use the best of the two worlds.

Our comparison is structured along two dimensions: (1) the spatial pattern of extension (homogeneous or heterogeneous) and (2) the temporal pattern of extension (static or dynamic). We concentrate on the implementation mechanisms associated with these two programming paradigms. For simplicity and purpose of a focused discussion, we do not take software-development methodologies, tool support, type systems, or other kinds of language-specific mechanisms into account. For feature-oriented programming this means that a feature module encapsulates a collaboration of software artifacts (classes and class refinements, in our case) that are composed by superimposition. For aspect-oriented programming this means that an aspect is a class-like entity that contains additionally pointcuts, advice, and inter-type declarations.

6.3.1 Homogeneous and Heterogeneous Crosscutting Concerns

Overview

We distinguish homogeneous and heterogeneous crosscutting concerns depending on the uniformity of the program extensions they induce.

Definition 6.10 A *homogeneous crosscutting concern* extends a program at multiple join points by applying a single *extension*. In contrast, a *heterogeneous crosscutting concern* extends multiple join points by adding multiple extensions, where each individual extension differs and targets exactly one join point.

(Colyer et al. 2004b) □

Extensions are either additions of new program elements (for example, implemented by inter-type declarations) or modifications or existing behavior (for example, implemented by advice).

For example, feature Colored is a homogeneous crosscutting concern. It extends the two classes Node and Edge in the same way (see Fig. 6.12): Aspect Colored advises two method executions (print in Node and Edge) and four inter-type declarations that introduce members and an interface to both classes, Node and Edge. In contrast, feature Weighted is a heterogeneous crosscutting concern (see Fig. 6.4). It extends the classes Graph and Edge but each in a different way: the extension of Graph introduces a method add; the extension of Edge introduces a method setWeight and a field weight, and it overrides method print.

In Fig. 6.18, we illustrate the difference between homogeneous and heterogeneous concerns. White boxes denote program elements, either elements being extended or elements that extend others (for example, advice or class refinements). Gray boxes denote the program and the crosscutting concern that extends the program. Note that a homogeneous crosscutting concern can be implemented using a set of distinct extensions, like a heterogeneous crosscutting concern; but this results in code replication. In Fig. 6.19, we depict an aspect with one piece of advice that advises three methods; in Fig. 6.20, we depict an equivalent aspect but with three distinct pieces of advice that advise only one method execution each—all with an identical advice body.

Comparison

Collaborations of classes are typically of a heterogeneous structure. That is, the roles and classes added to a program differ in their functionality, as in our graph example. A collaboration is a heterogeneous crosscutting concern, and a heterogeneous crosscutting concern can be understood as collaboration applied to a program. Feature modules are designed to implement heterogeneous crosscutting concerns.

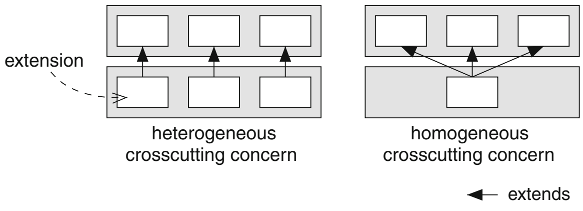

extension
heterogeneous crosscutting concern
homogeneous crosscutting concern
extends

Fig. 6.18 Homogeneous versus heterogeneous crosscutting concerns

Fig. 6.19 A homogeneous crosscutting concern implemented using a piece of advice

```
aspect FooAspect {
  after() : execution(void A.foo()) ||
            execution(void B.foo()) ||
            execution(void C.foo()) {
    /* do something */
  }
}
```

Fig. 6.20 A homogeneous crosscutting concern implemented using three pieces of advice

```
aspect FooAspect {
  after() : execution(void A.foo()) {
    /* do something */
  }
  after() : execution(void B.foo()) {
    /* do something */
  }
  after() : execution(void C.foo()) {
    /* do something */
  }
}
```

In contrast, aspects perform well in extending a set of join points using a single piece of advice, thus modularizing a homogeneous crosscutting concern, thereby avoiding code replication. The more join points are captured by a homogeneous crosscutting concern, the higher is the pay-off in terms of reduction of code replication. Although both feature-oriented and aspect-oriented programming support the crosscutting concerns the other focuses on, they cannot do both elegantly (Mezini and Ostermann 2004; Apel et al. 2008b).

For example, implementing feature Colored (a homogeneous crosscutting concern) using feature-oriented programming, we would apply two refinements to the classes Node and Edge, which introduce exactly the same code (see Fig. 6.21). Our aspect-oriented solution discussed previously avoids this code replication (see Fig. 6.22).

Conversely, an aspect may implement a collaboration (a heterogeneous crosscutting concern) by bundling a set of inter-type declarations and pieces of advice, as shown in Fig. 6.23; but, this way, the object-oriented structure of the program is lost and aspect-oriented mechanisms induce certain linguistic overhead (for example, to capture and pass arguments from pointcut to advice).

For this simple example, it does not matter whether one uses feature modules or aspects. The difference between paradigms becomes apparent when considering

```
refines class Node {
  Color color;
  Color getColor() { return color; }
  void print() {
    Color.setDisplayColor(getColor());
    Super.print();
  }
}
refines class Edge {
  Color color;
  Color getColor() { return color; }
  void print() {
    Color.setDisplayColor(getColor());
    Super.print();
  }
}
class Color { /* ... */ }
```

Fig. 6.21 Implementing feature Colored with feature-oriented programming

```
aspect Colored {
  interface IColored { Color getColor(); }
  declare parents: (Node || Edge) implements IColored;
  Color IColored.color;
  Color IColored.getColor() { return color; }
  before(IColored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.getColor());
  }
  static class Color { /* ... */ }
}
```

Fig. 6.22 Implementing feature Colored with aspect-oriented programming

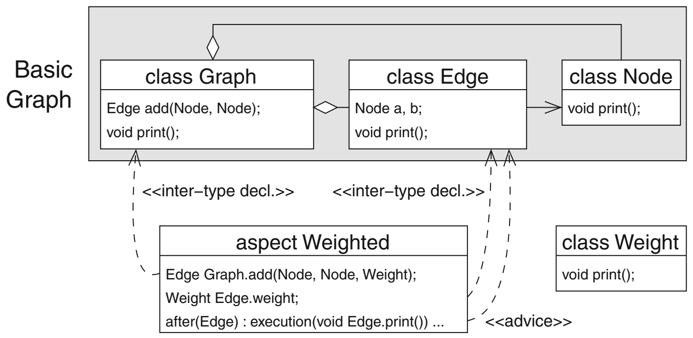


Fig. 6.23 Implementing a collaboration as an aspect

features at a larger scale. Suppose a base program consists of many classes and a feature extends most of them. In a feature-oriented solution the programmer defines a new role per class to be extended (see Fig. 6.25). This way, the programmer is able to recognize the program structure within the new feature. There is a one-to-one mapping between the structural elements of the base program and the elements of the feature (Fig. 6.24).

In an aspect-oriented solution, one would merge all participating roles into one aspect, as illustrated in Fig. 6.26. While this is possible, it flattens the inherent object-

```
aspect Weighted {
  Edge Graph.add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
  }
  Weight Edge.weight;
  after(Edge e) : this(e) && execution(void Edge.print()) {
    e.weight.print();
  }
}
```

Fig. 6.24 An AspectJ aspect that implements a collaboration

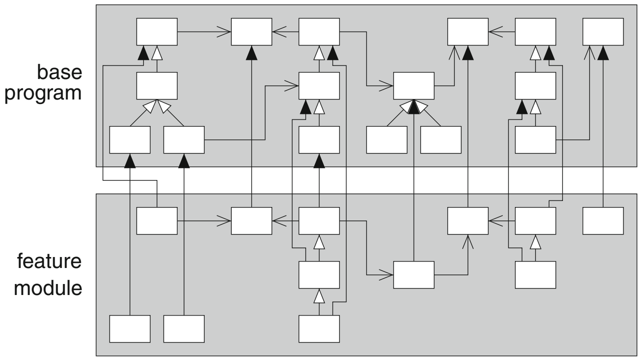


Fig. 6.25 Implementing a large-scale feature using a feature module (Apel 2007)

oriented structure of the feature (that is, the dominant decomposition; see Sect. 3.2.3) and makes it hard to trace the mapping between base program and feature (Steimann 2005; Mezini and Ostermann 2004; Apel et al. 2008b). The difference between feature-oriented and aspect-oriented solutions, as illustrated in Figs. 6.25 and 6.26, is not only a matter of visualization. The point is that the inner structure of the aspect does not reflect the structure of the base program; there is no natural mapping between structural elements of the base program and the feature. It is no coincidence that the mapping is complicated and hard to trace for the programmer. The one-to-one mapping of the feature-oriented solution is easier to understand especially for large-scale features.

Of course, implementing each role as a distinct aspect would be possible—in our example, we would implement the refinements of Graph and Edge as two distinct aspects. Doing so would enable to establish a one-to-one mapping between the structural elements of the base program and the elements of the feature. Bundled in a containment hierarchy, we could even locate all aspects contributing to a feature in a single place. In Sect. 6.3.4, we discuss this possibility—the combination of feature modules and aspects—but we also discuss why aspects should not be used this way.

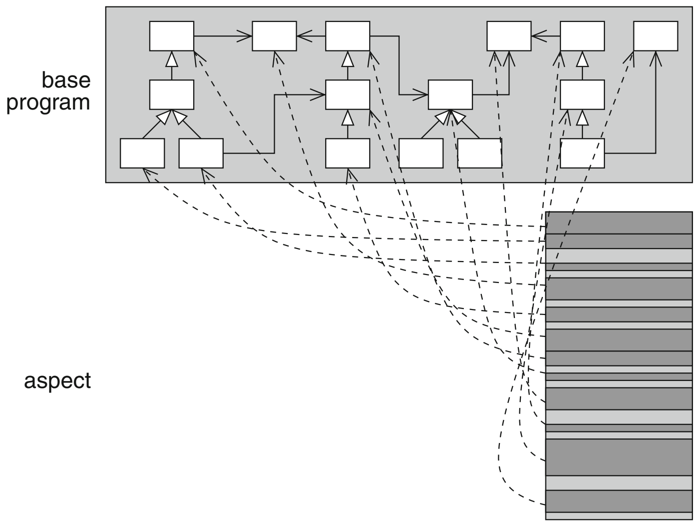


Fig. 6.26 Implementing a large-scale feature using an aspect (Apel 2007)

```
refines class Edge {
  Weight weight;
}
```

```
aspect Weighted {
  Weight Edge.weight;
}
```

Fig. 6.27 Implementing static crosscutting concerns in Jak (left) and AspectJ (right)

6.3.2 *Static and Dynamic Crosscutting Concerns*

Overview

A *static crosscutting concern* extends the structure of a program statically (Mezini and Ostermann 2004). That is, it adds new classes and interfaces and injects new fields, methods, interfaces, and so on. Note that method extensions are not static crosscutting concerns, as we explain shortly. AspectJ's inter-type declarations and Jak's class refinements that introduce new members are examples of static crosscutting concerns (see Fig. 6.27).

A *dynamic crosscutting concern* affects the control flow of a program (Mezini and Ostermann 2004). The semantics of a dynamic crosscutting concern can be understood and defined in terms of an event-based model (Wand et al. 2004; Lämmel 1999): A dynamic crosscutting concern runs additional code when predefined events occur during the program execution. Examples of programming constructs that implement dynamic crosscutting concerns are method extensions in Jak (via overriding) and advice in AspectJ (see Fig. 6.28).

The rule of thumb is that static crosscutting affects the *static program structure* in terms of newly introduced classes, methods, and fields (in Fig. 6.27, we add a field

```
refines class Edge {
  void print() {
    Super.print();
    weight.print();
  }
  /* ... */
}
```

```
aspect Weighted {
  after(Edge e) :
    execution(void Edge.print()) && this(e) {
      e.weight.print();
  }
  /* ... */
}
```

Fig. 6.28 Implementing dynamic crosscutting concerns in Jak (left) and AspectJ (right)

```
aspect TraceWeightChanges {
  after(Edge e) : set(Weight Edge.weight) && this(e) &&
    cflow(execution(void TestSuite.run())) {
    System.out.println("Weight value changed for " + e.print());
  }
}
```

Fig. 6.29 Tracing changes of edge weights only within the control flow of a test run

to a class using a class refinement and an inter-type declaration), whereas dynamic crosscutting affects the *dynamic program behavior* in terms of execution paths that are altered and added, and code that is executed additionally or instead of existing base code (for example, in Fig. 6.28, we execute an additional statement after the execution of an existing method). In other words, static crosscutting is concerned with program elements above the level of statements and expressions; dynamic crosscutting is concerned with the execution of statements and evaluation of expressions.

Dynamic crosscutting concerns are interesting when they not only affect method executions, but, for instance, field accesses, class instantiations, and the flow of exceptions. Work on aspect-oriented programming argues that expressing a program extension in terms of sophisticated events increases the abstraction level and captures the programmer's intention more directly (Mezini and Ostermann 2004). For example, in Fig. 6.29, we show an aspect that traces events in which weights of edges are changed, but only if they occur in the control flow of a test run (directly or indirectly invoked by TestSuite.run, for debugging purposes), not in an actual program execution. This control-flow dependency is expressed concisely using the predefined pointcut cflow.

Capturing and advising sophisticated events such as the one of Fig. 6.29 using classic implementation mechanisms, such as provided in Jak, results in complicated workarounds, as we illustrate shortly. There are many proposals for new language constructs for defining and catching new kinds of events during the program execution (Ostermann et al. 2005; Harbulot and Gurd 2006; Masuhara and Kawauchi 2003). In order to distinguish these new events and the novel language mechanisms that support them from simpler events known from object-oriented programming, we distinguish between *basic dynamic crosscutting concerns* and *advanced dynamic crosscutting concerns*:

```
refines class Edge
  implements Comparable {
    boolean compare(Edge e) {
      /* ... */
    }
}
```

```
aspect ComparableEdge {
  declare parents: Edge implements Comparable;
  boolean Edge.compare(Edge e) {
    /* ... */
  }
}
```

Fig. 6.30 Implementing a static crosscutting concern with class refinement (left) and inter-type declarations (right)

Definition 6.11 A *basic dynamic crosscutting concern* addresses only events that are related to method executions; *advanced dynamic crosscutting concerns* address all other events, for example, throwing an exception or assigning a value to a field. □

The definition implies that a basic dynamic crosscutting concern accesses only run-time variables that are related to the method execution that is advised, that is, arguments, result value, and enclosing object instance of the advised method; advanced dynamic crosscutting concerns can expose more information related to a join point, for example, the run-time type of the caller of a method. Furthermore, basic dynamic crosscutting concerns affect a program's control flow unconditionally, whereas advanced dynamic crosscutting concerns may specify a condition that is evaluated at run-time, for example, a method execution is only affected if it occurs in the control flow of another method execution. Finally, a basic dynamic crosscutting concern addresses only simple events; advanced dynamic crosscutting concerns can specify composite events and event patterns, for example, *trace matches* are executed when events fire in a specific pattern, thus, involving the history of computation (Allan et al. 2005).

A key property of basic dynamic crosscutting concerns is that they can be implemented as method extensions, as possible with inheritance and class refinement. They extend a method execution unconditionally and access only information that is available in method extensions, that is, the arguments, the result, and the enclosing run-time object. In short, basic dynamic crosscutting concerns can be used to implement collaborations.

Comparison

Both feature modules and aspects can extend the structure of a base program statically (that is, by injecting new members and introducing new superclasses and interfaces to existing classes). In Fig. 6.30, we depict a class refinement and an aspect, both of which inject a method and a field, and both of which introduce a new interface to class Edge.

Additionally, feature modules are able to encapsulate and introduce new classes (and interfaces), which is not possible with aspects (which can introduce only static

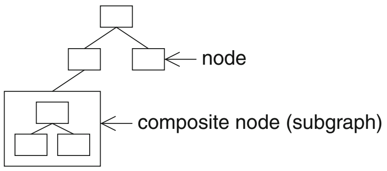


Fig. 6.31 A recursive graph data structure (Apel 2007)

```
aspect TraceGraphPrinting {
  before() : execution(void Graph.print()) &&
             !cflowbelow(execution(void Graph.print())) {
    System.out.println( /* ... */ );
  }
}
```

Fig. 6.32 Advising the printing mechanism using advanced advice

inner classes). In fact, AspectJ lacks a mechanism to group multiple aspects and classes that contribute to a feature to a larger composable unit (Lopez-Herrejon et al. 2005).

Feature-oriented programming provides no dedicated language support for implementing dynamic crosscutting concerns. Dynamic crosscutting concerns can be implemented—typically, only with preplanning and invasive changes—but there are no dedicated abstraction mechanisms to express them; feature-oriented programming supports only basic dynamic crosscutting concerns in the form of method extensions. In contrast, aspects provide a sophisticated set of mechanisms to refine a base program based upon its dynamic execution (for example, mechanisms for tracing the dynamic control flow and for accessing the run-time context of join points).

As an example, suppose we extend our graph example, such that a node may contain an inner graph, to model composite nodes, as illustrated in Fig. 6.31. Printing such an extended graph means that the Graph object invokes print on all of its Node objects, and that every Node object invokes print on its inner Graph object (if there is one), and so on.

Now, suppose we want to trace when a Graph object is printed, but only for the entire top-level graph, not its individual subgraphs. In order not to advise all executions of print, but only the top-level calls (that is, calls that do not occur in the dynamic control flow of other executions of print), we can use AspectJ's cflowbelow pointcut as condition evaluated at run time, as shown in Fig. 6.32; the advice implements an advanced dynamic crosscutting concern.

Generally, recursive data structures (for example, trees that contain subtrees that, again, contain subtrees, and so on) are an appropriate use case for aspect-oriented programming. Aspect-oriented language constructs for advanced dynamic crosscutting concerns (for example, cflow, cflowbelow) advise only selected join points within the control flow of a program. Though language abstractions such as cflow and cflowbelow can be implemented (emulated) by feature-oriented programming, this usually results in code replication, tangling, and scattering. For example, in Fig. 6.33,

```
 1 refines class Graph {
 2   static int count = 0;
 3   void print() {
 4     if (count == 0)
 5       System.out.println( /* ... */ );
 6     count++;
 7     Super.print();
 8     count--;
 9   }
10 }
```

Fig. 6.33 Implementing the extended printing mechanism with class refinement

we depict the above extension (Fig. 6.32) to the printing mechanism implemented using feature-oriented programming. Omitting aspect-oriented language constructs results in a complicated workaround (highlighted) for tracing the control flow (Lines 2,6,8) and executing the actual extension conditionally (Lines 4-5). Compared to the feature-oriented solution (which is even simplified and does not work with multiple Graph objects), the aspect-oriented solution captures the intension of the programmer more precisely and explicitly (see Fig. 6.32).

6.3.3 Summary of Comparison

In Table 6.1, we summarize the overall comparison. Feature-oriented programming focuses on collaborations, which are heterogeneous crosscutting concerns; aspect-oriented programming focuses on homogeneous crosscutting concerns, thus avoiding code replication. Furthermore, aspect-oriented programming is strong in abstracting the dynamic control flow, while feature-oriented programming's strength lies in abstracting collaborations that implement features. The benefits of using both feature-oriented and aspect-oriented programming together offer rewards that neither of them could accomplish in isolation. In the same vein, a feature may involve a mix of homogeneous and heterogeneous as well as of static and dynamic crosscutting, so a combination of different specialized mechanisms to handle them seems promising.

6.3.4 Combining Aspects and Feature Modules

As shown in Table 6.1, aspects and feature modules have mutual strengths and weaknesses, which suggests a combination of the two. The combination, as illustrated in Fig. 6.34, shows that the two are not competing approaches, but decompose in concert a program along three dimensions: classes, features, and aspects: an object-oriented design is the basis; aspects implement certain kinds of concerns that crosscut the underlying object-oriented design; feature modules decompose the design to impose

Table 6.1 A comparison of feature-oriented programming and aspect-oriented programming

	Feature-oriented programming	Aspect-oriented programming
Heterogeneous	*Good support*: Feature modules encapsulate and compose collaborations of classes and refinements	*Limited support*: Aspects bundle sets of inter-type declarations and advice, but obfuscate the object-oriented structure
Homogeneous	*No support*: Feature modules provide no explicit language constructs for refining multiple join points simultaneously	*Good support*: Aspects provide wildcards and pattern matching mechanisms to refine multiple join points simultaneously
Static	*Good support*: Feature modules can inject new fields, methods, classes, and interfaces, as well as declare new superclasses/interfaces	*Limited support*: Aspects can inject new fields and methods as well as declare new superclasses/interfaces, but cannot inject classes or interfaces
Dynamic	*Weak support*: Feature modules can implement only basic dynamic crosscutting concerns via overriding (method extensions); there is no support for advanced dynamic crosscutting concerns	*Good support*: Aspects provide sophisticated mechanisms for advising a program based on its dynamic execution (basic and advanced dynamic crosscutting concerns)

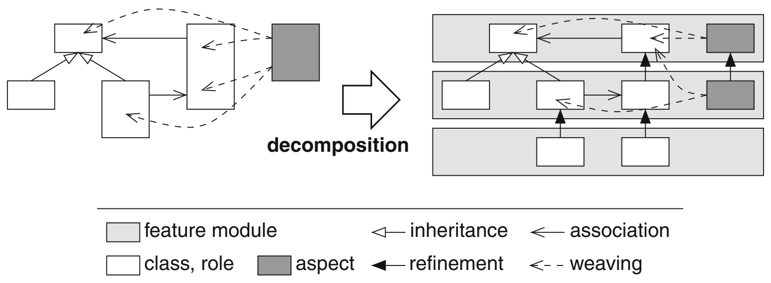


Fig. 6.34 Feature-driven decomposition of an aspect-oriented design (Apel et al. 2008b)

a structure that is of interest to stakeholders, that is, the features of a program. Hence, a feature is implemented by a collaboration of classes *and* aspects (right side of Fig. 6.34).[5] In this symbiosis, features and aspects profit from each other and overcome their individual limitations.

Feature modules that contain aspects are called henceforth *aspectual feature modules*. In Fig. 6.34, the aspectual feature module (middle layer) refines the base program (top layer) in two ways: (1) it adds two classes and a class refinement, and (2)

[5] Note that the original aspect has been split into two pieces (a base aspect and a subsequent refinement), which is discussed elsewhere (Apel et al. 2007).

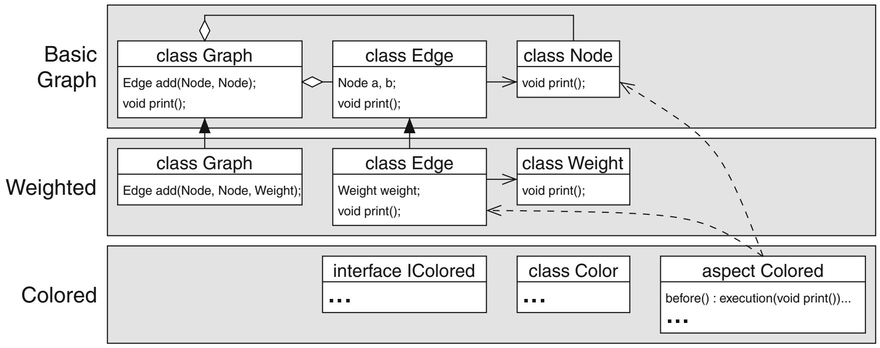


Fig. 6.35 Implementing feature Colored as an aspectual feature module (Apel et al. 2008b)

it weaves an aspect to implement crosscutting changes. Probably the most important contribution of aspectual feature modules is that programmers may choose the appropriate technique, refinements or aspects, that fits a given problem best.

In Fig. 6.35, we depict the collaboration-based design of our graph example, consisting of the features BasicGraph, Weighted, and Colored. Feature Colored is implemented by means of an aspect, a class, and an interface; all are encapsulated by an aspectual feature module. As discussed earlier, advising executions of the print methods in Node and Edge is a homogeneous crosscutting concern—the same is true for injecting field color and the methods setColor and getColor to Node and Edge (see Fig. 6.22). In this situation, it is beneficial to use an aspect because it is able to avoid replicated code. Encapsulating aspect Colored, interface IColored, and class Color improves feature cohesion and traceability, compared to a pure aspect-oriented implementation.

As with standard feature modules, an aspectual feature module is represented as a containment hierarchy. Beside Jak files, an aspectual feature module also contains aspect files. In Fig. 6.36, we depict the simplified containment hierarchies of our graph features BasicGraph, Weighted, and Colored. The containment hierarchy synthesized is generated by composing the three feature hierarchies. The result is a set of collaborating software artifacts. In the case of aspectual feature modules, we have an aspect-oriented program (and in the case of traditional feature modules, an object-oriented program). Now it becomes clear that it is necessary to weave the aspects and the object-oriented base program in a subsequent step, after all classes and their refinements have been composed. These two steps can be accomplished by different compiler passes or by different tools (Apel 2007).

6.3.5 A Study on Advanced Crosscutting Mechanisms

In the previous sections, we discussed the mutual strengths and weaknesses of feature-oriented and aspect-oriented programming, and we discussed an approach to combine

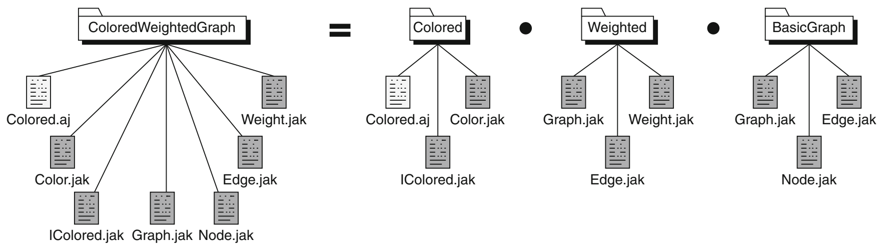


Fig. 6.36 Composing containment hierarchies that include aspects

both. While feature-oriented programming à la Jak is a modest extension to object-oriented programming, aspect-oriented programming à la AspectJ introduces a number of novel and powerful mechanisms for the implementation of homogeneous and advanced dynamic crosscutting concerns. The downside of this expressiveness is that mechanisms such as advice and quantification may actually violate the principle of information hiding and hamper program comprehension, maintenance, and evolution (see Sect. 6.2.4). So, questions arise on how frequently these advanced crosscutting mechanisms are actually used, as compared to basic mechanisms that are also directly supported in feature-oriented programming?

The first author (Apel) has conducted an empirical study to investigate how programmers use AspectJ and whether they use the advanced and unique mechanisms provided by the language (Apel 2010). Apel defined a number of source-code metrics to quantify the usage of individual mechanisms provided by AspectJ, including basic and advanced crosscutting mechanisms. The study was based on a diverse selection of publicly available AspectJ programs from different domains and of different sizes (up to 130,000 lines of code); a summary of the results is shown in Table 6.2.

The key result of this study is that only a minor fraction of extensions use advanced crosscutting mechanisms. In Fig. 6.37, we depict the fractions of the code base of the AspectJ programs that exploit basic and advanced crosscutting mechanisms (which are also supported by feature-oriented programming). On average, only $2 \pm 2\,\%$ of the analyzed code exploits the advanced capabilities of AspectJ; $12 \pm 9\,\%$ implements basic aspects, and the remaining $86\,\%$ is object-oriented code.

Furthermore, the use of aspects led to a reduction of code size of $6 \pm 9\,\%$, which is in line with prior work on clone detection (Baxter et al. 1998), which estimates that 5 to 15 % of large software projects are clones. So, there might be an untapped potential (further 9 %=15–6 %) of AspectJ to reduce code replication further because not all clones have been discovered.

6.3.6 Discussion

Aspectual feature modules are independent of a specific host language. They can be implemented in any pair of object-oriented and aspect-oriented language that

Table 6.2 Overview of the AspectJ programs analyzed by Apel (2010)

Name	LOC	Source	Description
Tetris	1,030	Blekinge Inst. of Technology[a]	Implementation of the popular game
OAS	1,623	Lancaster University[b]	Online auction system
Prevayler	3,964	University of Toronto[c]	Main-memory database system
AODP	3,995	University of British Columbia[d]	AspectJ impl. of 23 design patterns
FACET	6,364	Washington University[e]	CORBA event-channel implementation
ActiveAspect	6,664	University of British Columbia[f]	Crosscutting-structure presentation tool
HealthWatcher	6,949	Lancaster University[g]	Web-based information system
AJHotDraw	22,104	open source project[h]	2D graphics framework
Hypercast	67,260	University of Virginia[i]	Protocol for multicast overlay networks
AJHSQLDB	75,556	University of Passau[j]	SQL relational database engine
Abacus	129,897	University of Toronto[k]	CORBA middleware framework

[a]http://www.guzzzt.com/coding/aspecttetris.shtml
[b]The sources were kindly released by A. Rashid.
[c]http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem/
[d]http://www.cs.ubc.ca/~jan/AODPs/
[e]http://www.cs.wustl.edu/~doc/RandD/PCES/facet/
[f]The sources were kindly released by W. Coelho and G. Murphy.
[g]The sources were kindly released by A. Garcia.
[h]http://sourceforge.net/projects/ajhotdraw/
[i]The sources were kindly released by Y. Song and K. Sullivan.
[j]http://sourceforge.net/projects/ajhsqldb/
[k]The sources were kindly released by C. Zhang and H.-A. Jacobsen.

can be woven, for example, Java and AspectJ, C++ and *AspectC++*[6], or C# and *AspectC#*[7], and others. This circumstance makes the concept of aspectual feature modules invariant to the specifics of the host languages.

Including aspects in the layered organization of feature modules raises a number of further issues. Most importantly, there is the issue of the scope of aspects within the layer structure. There are two alternatives: (1) aspects may affect only program elements of layers above (add previously) and within the own feature module, or (2) aspects may affect all program elements, even those of feature modules added subsequently. While the former alternative enforces a disciplined incremental development style, the latter improve extensibility. For a deeper discussion, we refer the reader elsewhere (Apel et al. 2010).

[6] http://www.aspectc.org/
[7] http://www.dsg.cs.tcd.ie/dynamic/?category_id=169

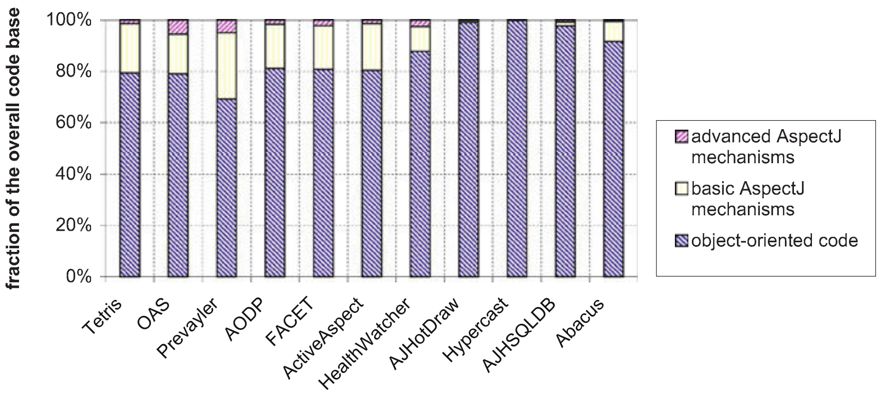


Fig. 6.37 Fractions of basic and advanced AspectJ mechanisms of the overall analyzed code base (Apel 2010)

Apel's study of 11 AspectJ programs revealed that only 2 % of the code base takes advantage of the sophisticated mechanisms provided by aspect-oriented programming à la AspectJ (Apel 2010). This raises the question of whether this fraction justifies the use of AspectJ, thereby inviting the fragile-pointcut problem and others (see Sect. 6.2.4). Or is this fraction relevant and justifies the use of AspectJ, especially as Apel found a noticable reduction in code size when using aspects? Certainly, a compromise is to use the basic crosscutting mechanisms of feature-oriented programming for heterogeneous and basic dynamic crosscutting concerns and the advanced crosscutting mechanisms of aspect-oriented languages (pointcuts and advice) for homogeneous and advanced dynamic crosscutting concerns. Aspectual feature modules are a promising approach that supports this combination.

6.4 Tooling

AHEAD[8] and *FeatureHouse*[9] are the most popular tools for feature-oriented programming. They are publicly available for experimentation including several examples. Both are command-line tools, used mainly for academic purposes. The tool *FeatureIDE* provides a graphical front-end in Eclipse, with corresponding editors, a mapping from features to feature modules, automatic composition of selected features in the background, generation of collaboration diagrams, and much more (see Appendix A). *FeatureIDE* ships with *AHEAD* and *FeatureHouse* and several example projects, ready to explore. After a developer graphically configures the desired features, *FeatureIDE* automatically invokes the corresponding composition tools such as the Jak compiler. It is likely the easiest way to try *AHEAD* and *FeatureHouse*, for

[8] http://www.cs.utexas.edu/users/schwartz/ATS.html

[9] http://fosd.net/fh

developers familiar with Eclipse. A video tutorial on *FeatureIDE* is available on the web.[10] In Appendix A, we get back to *FeatureIDE* and explain its facilities in more detail.

AspectJ is the most-popular and widely-used aspect-oriented language.[11] It is publicly available, can be used out of the box, and it is deployed with proper documentation, tutorial, and examples. Beside *AspectJ*, there are several aspect-oriented languages available including *AspectC++*[12] for C++ and *ACC*[13] for C. *AspectJ* and *AspectC++* are supported by plug-ins in Eclipse (*AJDT*[14] and *ACDT*[15]), which include syntax highlighting, enabling and disabling aspects selectively, and various visualization and navigation facilities. Furthermore, there are a number of framework-based solutions available that incorporate concepts from aspect-oriented programming, such as *JBoss AOP*[16] and *Spring*.[17] Finally, *FeatureIDE* supports the development of feature-oriented product lines by integrating *AJDT* into the product-line–development life cycle.

FeatureC++[18] is a language extension of C++ that supports feature-oriented programming (Apel et al. 2005). It consists of a tool for composing feature modules and a compiler for C++ artifacts. *FeatureC++* supports aspectual feature modules by integrating *AspectC++* (Spinczyk et al. 2005). A further way to implement aspectual feature modules is to combine the *AHEAD* tool suite and *AspectJ*. While Jak is used to compose traditional feature modules, AspectJ weaves the aspects of the feature modules into the synthesized class hierarchies (Apel 2007). However, this combination does not work "out of the box" and requires manual tool integration.

6.5 Practical Relevance

Finally, we would like to say a few words on the practical relevance of aspect-oriented and feature-oriented programming, as well as on ways of industrial adoption. Clearly, work on aspect-oriented and feature-oriented programming is still mainly driven by academics. A major goal of academic research is to explore and understand why classic implementation approaches often fail to modularize code. Work on aspect-oriented programming introduced the concept of crosscutting concerns as well as mechanisms to implement them. Work on feature-oriented programming emphasized

[10] http://www.cs.utexas.edu/users/dsb/cs392f/Videos/FeatureIDE/

[11] http://www.eclipse.org/aspectj/

[12] http://www.aspectc.org/

[13] http://research.msrg.utoronto.ca/ACC

[14] http://www.eclipse.org/ajdt/

[15] http://acdt.aspectc.org/

[16] http://www.jboss.org/jbossaop

[17] http://www.springsource.org/

[18] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

feature modularity, feature composition, and proper handling of feature interactions (see Chap. 9).

Research on aspect-oriented and feature-oriented programming made significant progress, but corresponding languages are mainly used for implementing academic tools and conducting empirical studies. Although there are several substantial systems written using these implementation approaches (for example, the AHEAD tool suite (Batory et al. 2004) and the IBM middleware product line (Colyer and Clement 2004)), there remains a long way to industrial adoption. While it is common that results from basic research need their time to ripe, there are other more subtle ways for the transition of ideas of aspect-oriented and feature-oriented programming to industrial practice. A notable example is the inclusion of noninvasive extension mechanisms such as traits and partial classes—both closely related to Jak's class refinement—to mainstream languages such as Scala and C#. Another example is the inclusion of aspect-oriented concepts in major frameworks such as Spring (Dessi 2009).

6.6 Further Approaches

A number of programming mechanisms has been proposed to support the implementation of roles and collaborations, most notably, virtual classes (Madsen and Moller-Pedersen 1989), classboxes (Bergel et al. 2005), mixin layers (Smaragdakis and Batory 2002), and object teams (Herrmann 2002). Although they differ in details from feature modules (Apel 2010), the big picture and the basic concepts are the same, especially, with regard to the comparison of feature-oriented programming and aspect-oriented programming (see Sect. 6.3.3). An in-depth discussion of their individual differences is outside the scope of this book. Next, we concentrate on programming paradigms that are in the spirit of aspect-oriented and feature-oriented programming, but diverge substantially from the core concepts of the two. We outline the most important approaches, from our perspective.

6.6.1 Delta-Oriented Programming

The idea of *delta-oriented programming* is closely related to aspect-oriented and feature-oriented programming (Schaefer et al. 2010). A program consists of a base module and a set of delta modules that modify the base module in a stepwise manner. This asymmetry between the two kinds of modules is similar to aspect-oriented programming, which distinguishes between base program and aspects, and to feature-oriented programming, which distinguishes between classes and class refinements. Much like a feature module, a delta module can add new classes and members as well as extend existing methods by overriding. In contrast to feature modules, delta modules can also delete existing classes and individual members. In Fig. 6.38, we

```
core BasicGraph {
  class Graph { /*...*/ }
  class Node {
    int id = 0;
    Node(int _id) { id = _id; }
    void print() { System.out.print(id); }
  }
  class Edge { /*...*/ }
}
```

```
delta IdToName when BasicGraph {
  modifies class Node {
    removes void print();
    adds void display() { System.out.print(id); }
  }
}
```

Fig. 6.38 Using a delta module to replace method print with method display

use a delta module to replace method print of our graph example (which is contained in the base module) with another method with name display.

The fact that delta modules can delete program elements has several implications. First, delta modules may shrink the base program; with feature modules, it always grows. That is, one can start with a monolithic program and incrementally remove functionality that is associated with individual features, which is similar to annotative approaches such as using preprocessors and conditional compilation (see Sect. 5.3), and which aligns well with the extractive model of product-line engineering (see Sect. 2.4.2).

An issue is that the additional expressiveness gained by using delta modules may cause problems regarding maintenance and program comprehension. For example, looking at a program, it may be hard to overview whether some other delta module will delete an element that is needed. Work on type checking of delta-oriented programs aims at alleviating this problem (Schaefer et al. 2011).

6.6.2 *Refactoring Feature Modules*

A similar approach but stricter than delta modules in terms of permitted transformations is *refactoring feature modules* (Kuhlemann et al. 2009a). The idea is to include refactorings (that is, declarative descriptions of refactorings to perform) beside classes and class refinements into a feature module. As refactorings change the structure but not the observable behavior, the expressiveness is limited compared to delta modules. A typical use case is to adapt a given feature module to a base program or library, which requires guarantees that the adapter does not alter the overall program behavior.

In Fig. 6.39, we use the refactoring feature module NodeToVertex to rename class Node of feature BasicGraph to Vertex, so that it become compatible with the expecta-

```
layer BasicGraph;

class Graph { /*...*/ }
class Node { /* ... */ }
class Edge { /* ... */ }
```

```
layer NodeToVertex

refactoring NodeToVertex implements RenameClassRefactoring {
  String getOldClassId() { return "Node"; }
  String getNewClassName() { return "Vertex"; }
}
```

```
layer MinSpanTree;

class MinSpanTree {
  Vertex compute(Vertex start) { /* ... */ }
}
```

Fig. 6.39 Using a refactoring feature module for renaming class Node to Vertex, in order to make it compatible with the implementation of feature MinSpanTree

tions of feature MinSpanTree. Similar to delta-oriented programming, operations such as renaming are potential sources of confusion and errors, which can be alleviated by proper analysis mechanisms (Kuhlemann et al. 2009b).

6.6.3 Context-Oriented Programming

The basic idea of *context-oriented programming* is that the programming language should support the programmer to express context-dependent behavior of a program (Hirschfeld et al. 2008). That is, depending on the current context, which may change dynamically at run-time, program behavior can differ. Context is every piece of information that is external to the program such as the operating system, network throughput, physical location of the device, and so on. Hence, context-oriented programming is a language-based approach that is related to run-time adaptation, location-aware services, and dynamic software evolution.

The relation to product lines is as follows: One can view the behaviors that are triggered in certain contexts as features of the system, the triggering mechanism as dynamic feature composition (Rosenmüller et al. 2011), and the family of systems that emerge from different contexts that occur in a domain as a product line. Technically, context-oriented languages use a variety of mechanisms to implement the different context-dependent behaviors including collaborations and roles as well as specific dynamic or static conditions based on pointcut-like mechanisms (Hirschfeld et al. 2008).

In Fig. 6.40, we show an implementation of feature Colored using a context-dependent layer written in the context-oriented language *ContextJ* (Appeltauer et al.

```
class Node {
  int id = 0;
  Node(int _id) { id = _id; }
  void print() { /* ... */ }
  layer Colored {
    Color color = new Color();
    Color getColor() { return color; }
    before void print() {
      Color.setDisplayColor(getColor());
    }
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() { /* ... */ }
  layer Colored {
    Color color = new Color();
    Color getColor() { return color; }
    before void print() {
      Color.setDisplayColor(getColor());
    }
  }
}
```

Fig. 6.40 Implementing feature Colored as a context-dependent layer in ContextJ

2011). Using ContextJ's layer activation mechanism, one can activate and deactivate feature Colored at run time. Note that the implementation of Colored is scattered across the classes Node and Edge: the code that belongs to a layer within a given class is declared using keyword layer. ContextJ provides a mechanism similar to AspectJ's advice to extend method executions (for example, keyword before). In other context-oriented languages, all code that belongs to a layer (that is, a feature in our case) can be located in a single spot (Hirschfeld et al. 2008).

6.7 Further Reading

There are no principle text books on feature-oriented programming. Seminal research papers include the papers of Prehofer (1997) and Batory et al. (2004). The text book of Czarnecki and Eisenecker (2000) covers some material related to feature-oriented and aspect-oriented programming. Apel and Kästner (2009) provide a quite up-to-date overview of research on feature-oriented software development.

There are several text books about aspect-oriented programming and AspectJ (Laddad 2003; Colyer et al. 2004a; Gradecki and Lesiecki 2003; Dessi 2009). In particular, Laddad's book and the *AJDT* plug-in provide a good start to learn AspectJ, the major aspect-oriented language. For readers interested in original publications, we refer to Kiczales' papers introducing the ideas of aspect-oriented programming (Kiczales et al. 1997) and the AspectJ language (Kiczales et al. 2001). Filman et al. (2005) collect (early) key papers on aspect-oriented programming in a comprehensive text book. Steimann (2006) provides a comprehensive overview of core work on aspect-oriented programming, mostly from a critical perspective. Several researchers discuss the potential of aspect-oriented programming for developing product lines (Griss 2000; Lee et al. 2006; Mezini and Ostermann 2004; Lohmann et al. 2006b; Voelter and Groher 2007; Apel et al. 2008b).

The combination of feature-oriented and aspect-oriented programming is relatively new. The approach of aspectual feature modules is explained in detail by Apel et al. (2008b). Several researchers explored combinations of feature-oriented and aspect-oriented programming that are similar to (and even predate) the approach of aspectual feature modules, in particular, *CaesarJ* (Aracic et al. 2006), *Object Teams* (Herrmann 2002), and *Aspectual Collaborations* (Lieberherr et al. 2003).

Exercises

6.1. Explain and relate the following concepts:

(a) Feature and crosscutting concern,
(b) Aspect, role, and collaboration,
(c) Class refinement, inter-type declaration, and advice,
(d) Homogeneous and heterogeneous crosscutting concern, and
(e) Static and dynamic crosscutting concern.

6.2. Implement the chat system (Exercise 4.1, p. 96) using feature-oriented programming. If using Java, we recommend using *FeatureHouse* within the *FeatureIDE* plug-in or Jak in combination with the AHEAD tool suite (see AppendixA).

(a) Before implementation, design a collaboration diagram covering the key concepts and extensions.
(b) Reimplement the chat system using feature-oriented programming. Critically discuss separation of concerns, code quality, feature traceability, and effort. Update the collaboration diagram if necessary.
(c) Subsequently add an additional feature Sound: When sending and receiving messages, corresponding sounds should be played. Also, when typing a message, there should be a sound on every keystroke. Discuss effort and the degree of invasiveness of implementing this additional feature.

6.3. Find a project implemented with feature-oriented programming. Given the academic status of current feature-oriented languages, the best chance will be selecting a program from the *SPL2go* repository,[19] an example program bundled with *FeatureHouse* or the *AHEAD* tool suite, or one of the examples deployed with *FeatureIDE* (see Appendix A). Explore how feature-oriented concepts have been used in this program.

(a) Create a collaboration diagram based on the feature-oriented implementation.

[19] http://spl2go.cs.ovgu.de/

(b) What language constructs are used?
(c) Are extensions homogeneous or heterogeneous?
(d) What kind of concerns are separated into feature modules? Do feature modules represent product-line features?

6.4. Implement the chat system (Exercise 4.1, p. 96) using aspect-oriented programming. If using Java, we recommend using AspectJ with the AJDT and FeatureIDE's Eclipse plug-ins.

(a) Summarize which language constructs have been used in your implementation? Were there opportunities to exploit aspect-oriented mechanisms for homogeneous extensions?
(b) Critically discuss separation of concerns, code quality, and effort of the implementation.
(c) Subsequently add an additional feature BusyStatus: Users can set the status of their clients as available or busy; the status is sent to the server; the server withholds all messages from a busy user and sends all messages in a bulk update after the user becomes available again. Observe effort and the degree of invasiveness of implementing this additional feature.

6.5. Find an industrial or academic open-source project that uses aspect-oriented programming. Popular examples are AJHotdraw,[20] Health Watcher,[21] and MobileMedia.[22] Investigate how AspectJ has been used. What kind of concerns are implemented as aspects? What aspect-oriented language constructs are used? Are extensions homogeneous or heterogeneous? Are aspects used to implemented features in a product line?

6.6. Rewrite the three code fragments below as follows: First remove all highlighted source code; subsequently write (a) a class refinement with Jak and (b) an aspect with AspectJ that recreate the original behavior of the source code.

Discuss the expressive power of Jak and AspectJ. What kind of extensions are easily possible and what kind of extensions require workarounds?

```
public class Foo {
  void main(Lock lock) {
    int lockId = lock.lock();
    try {
      log("main");
    } finally {
      Lock.unlock(lockId);
    }
  }
}
```

```
public class Bar {
  void main(String p) {
    int i = p.length();
    firstOperation(i);
    log("first: " + i);
    secondOperation();
    thirdOperation();
  }
  void log(String m) { ... }
}
```

```
public class Pizza {
  void main(String p) {
    int i = p.length();
    int x = i * 2;
    x = x + rand(i);
    x++;
    System.out.println(x);
  }
  int rand(int i) { ... }
}
```

6.7. In Fig. 4.9 (p. 78), we discussed limitations of inheritance. Does feature-oriented programming or aspect-oriented programming overcome this limitation? Illustrate your argument with the example of Fig. 4.9.

[20] http://sourceforge.net/projects/ajhotdraw/
[21] http://www.comp.lancs.ac.uk/~greenwop/tao/
[22] http://mobilemedia.sourceforge.net/

6.8. Consider implementing a stack class with two optional features Logging and Synchronization. Discuss the difference between inheritance (**class** SyncStack **extends** Stack { ... }), the decorator pattern (**class** SyncStack {SyncStack(Stack s)... }), feature modules (**layer** Sync; **class** Stack { ... }) and aspects (**aspect** Sync { ... }) with regard to preplanning. What form of preplanning is required in each implementation strategy?
6.9. Obliviousness in aspect-oriented programming is a controversial concept.

(a) Discuss benefits and drawbacks of obliviousness, especially with regard to preplanning and information hiding. Can you give examples from your implementations (Exercise 6.4) or open-source projects (Exercise 6.5) to support your statements?
(b) Does the concept of obliviousness also apply to feature-oriented programming? Discuss the differences and commonalities of aspect-oriented and feature-oriented programming with regard to obliviousness.

6.10. Select a code or noncode language for which there is not already feature-oriented or aspect-oriented support (for example, JavaScript, Scheme, Make, SQL, or Markdown). Sketch how a feature-oriented or aspect-oriented extension of this language could look like. Give an example of a code example in that language and two optional features of your choice extending the code.
6.11. Discuss the mutual strengths and weaknesses of feature-oriented and aspect-oriented programming based on the Jak and AspectJ implementations of the chat system (Exercises 6.2 and 6.4).
6.12. Can you identify synergies by combining feature-oriented and aspect-oriented programming when implementing the chat system (Exercise 4.1, p. 96)? Revise the implementation of Exercise 6.2 accordingly. Follow the guidelines of Table 6.1. Discuss what you gained by combining the two approaches, if anything.
6.13. Based on your experience and the discussions so far, judge the practical relevance of programming-language mechanisms that support:

(a) homogeneous crosscutting concerns,
(b) heterogeneous crosscutting concerns,
(c) static crosscutting concerns,
(d) basic dynamic crosscutting concerns, and
(e) advanced dynamic crosscutting concerns.

6.14. Discuss the benefits and drawbacks of using the alternative implementation approaches discussed in Sect. 6, compared to feature-oriented and aspect-oriented programming. Refer to your chat implementation or code fragments of open-source systems to support your points.
6.15. Reconsider the scenarios of Exercise 2.9 (p. 44). Which implementation approach would you recommend to the developers and why? Under which conditions would you recommend feature-oriented or aspect-oriented programming?
6.16. Extend the comparison of Exercise 4.11 (p. 97) with the additional implementation strategies from this chapter.

Chapter 7
Advanced, Tool-Driven Variability Mechanisms

After reading the chapter, you should be able to

- discuss use cases of leveraging traceability information beyond variability in product generation,
- tradeoff benefits and drawbacks of views on source code compared to compositional implementations (that is, virtual versus physical separation of concerns),
- discuss opportunities for future tool-driven variability mechanisms, and for the integration of development phases.

There are several attempts to support the development and management of feature-oriented product lines by means of tool support that exceeds traditional tools such as preprocessors, build systems, and version control systems. They typically build on concepts of build systems and conditional compilation, but provide tool support that goes beyond traditional systems. Most tool-driven solutions are available only in academic prototypes yet. We introduce three classes of tools that build on one another in Sects. 7.1–7.3, and discuss their strengths and weaknesses in Sect. 7.4.

7.1 Exploiting Feature Tracing

An idea to alleviate the problem of crosscutting feature implementations is to leverage and maintain *tracing links* between the features of the feature model and the artifacts implementing the features.

Definition 7.1 A *tracing link* connects a feature with its implementation artifacts (or parts thereof). □

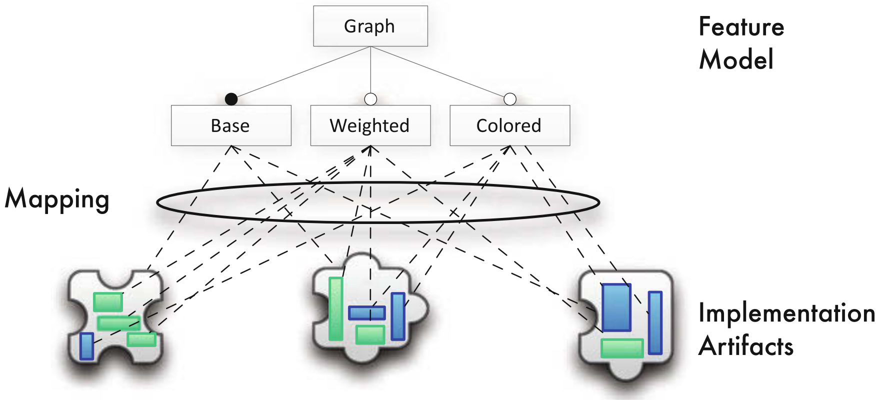
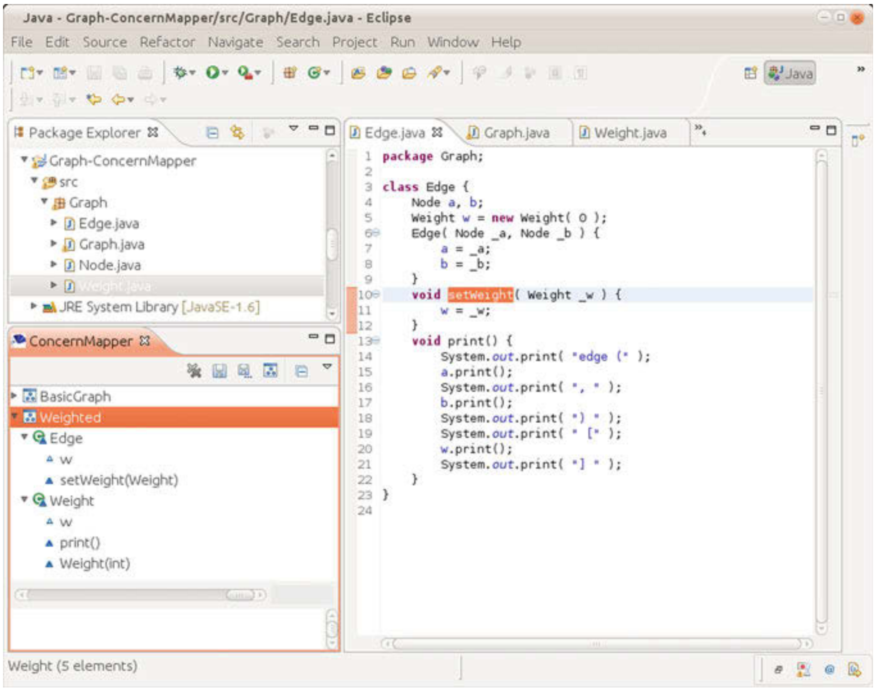


Fig. 7.1 Tool-managed mapping between features and implementation artifacts using tracing links (dashed lines)

Typically, tracing information is available during development; the key is to capture and exploit this information. Proper tool support can help to maintain the mapping between each individual feature and its program elements, as illustrated in Fig. 7.1, and present it to developers for given maintenance tasks. Programmers can modify the feature model, the implementation artifacts, as well as the mapping between the two by assigning explicit tracing links.

Depending on the feature implementation technique, tracing information is more or less easy to capture and maintain, as discussed in the previous chapters. For example, using frameworks or feature-oriented programming (see Sects. 4.3 and 6.1, pp. 79 and 130), we can trace a feature to its implementation in a single module (often feature and module share the same name). When using preprocessor directives (see Sect. 5.3, p. 110), we can reconstruct tracing links from features to multiple code locations, given the consistency assumption that feature names and corresponding preprocessor constants match. However, using runtime parameters in the program or build system, tracing is much more challenging, because it is difficult to statically determine which code fragments the parameter affects (see Sects. 4.1 and 5.2, pp. 66 and 105). For yet other concerns of the system that are currently not configurable, no tracing information may be present in the source code; in such cases, tracing links can be extracted manually or with semi-automatic feature-location tools (Biggerstaff et al. 1993; Robillard and Murphy 2002; Poshyvanyk et al. 2007; Cornelissen et al. 2009).

Tracing links cannot only be extracted from implementations (such as preprocessor directives or feature-module names), they can be managed as prime development artifact in a tool. Let us illustrate feature tracing by means of our graph example. Given just plain Java code without variability, we can use a separate tool to trace features BasicGraph and Weighted, as shown in Fig. 7.2 with the tool *ConcernMapper* (Robillard and Weigand-Warr 2005). The tool maintains a separate mapping from a list of concerns (or features) to code fragments. Several other tools have been



```
package Graph;

class Edge {
    Node a, b;
    Weight w = new Weight( 0 );
    Edge( Node _a, Node _b ) {
        a = _a;
        b = _b;
    }
    void setWeight( Weight _w ) {
        w = _w;
    }
    void print() {
        System.out.print( "edge (" );
        a.print();
        System.out.print( ", " );
        b.print();
        System.out.print( ") " );
        System.out.print( " [" );
        w.print();
        System.out.print( "] " );
    }
}
```

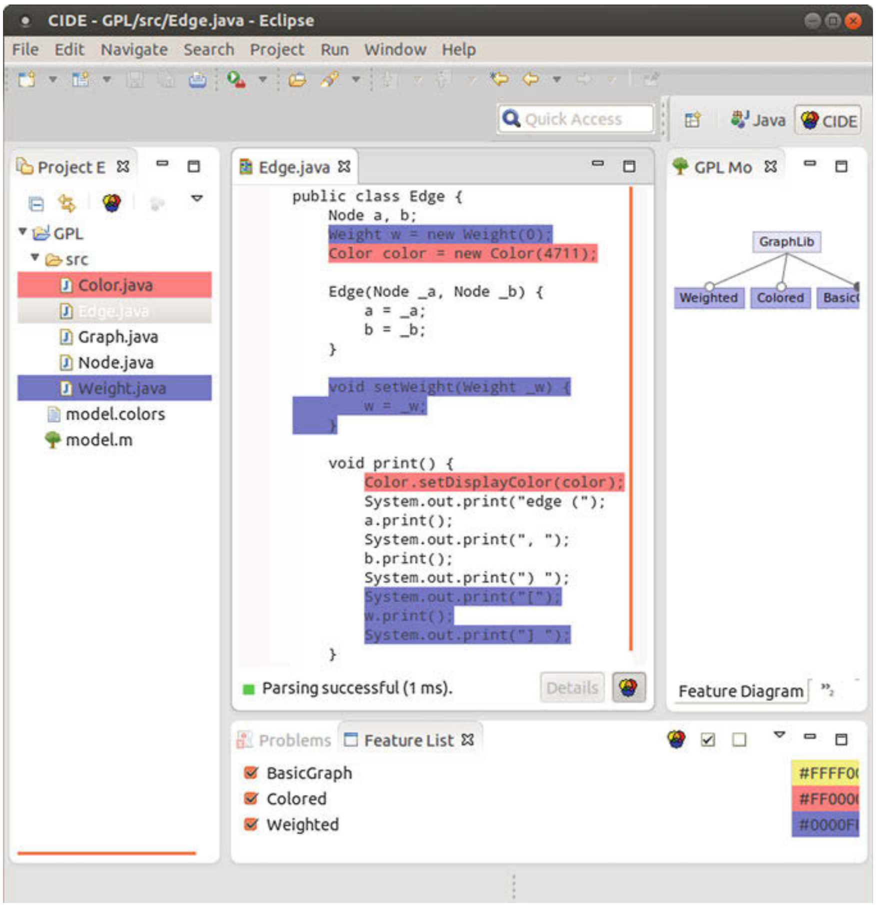
Fig. 7.2 Feature tracing with the tool ConcernMapper. Concerns and their associated code structures are shown in the bottom left view

developed to maintain similar tracing links, including *CIDE* (Kästner et al. 2008a), *fmp2rms* (Czarnecki and Antkiewicz 2005), and *FeatureMapper* (Heidenreich et al. 2008b)—the last two map features to fragments of graphical models. In all cases, the internal mapping between feature and artifact fragments is maintained in a mapping model, but the code and model artifacts are not actually changed. Since the mappings are based on the artifacts' structure, the mapping is similar to disciplined (syntactic) preprocessor annotations (see Sect. 5.3, p. 110). If changes are made under the control of the tracing tool, the tool can update the map to preserve the mapping, for example, when moving code to a different file.

Having tracing links accessible to tools (either extracted from implementations or directly specified by the programmer) has benefits beyond just easing the implementation of variability: consistency checking and visualization.

7.1.1 Consistency Checking

We can check or enforce *consistency* of the mapping. For example, Tartler et al. (2011) found several preprocessor directives in the Linux kernel that refer to undefined



```
public class Edge {
    Node a, b;
    Weight w = new Weight(0);
    Color color = new Color(4711);

    Edge(Node _a, Node _b) {
        a = _a;
        b = _b;
    }

    void setWeight(Weight _w) {
        w = _w;
    }

    void print() {
        Color.setDisplayColor(color);
        System.out.print("edge (");
        a.print();
        System.out.print(", ");
        b.print();
        System.out.print(") ");
        System.out.print("[");
        w.print();
        System.out.print("] ");
    }
```

Parsing successful (1 ms).

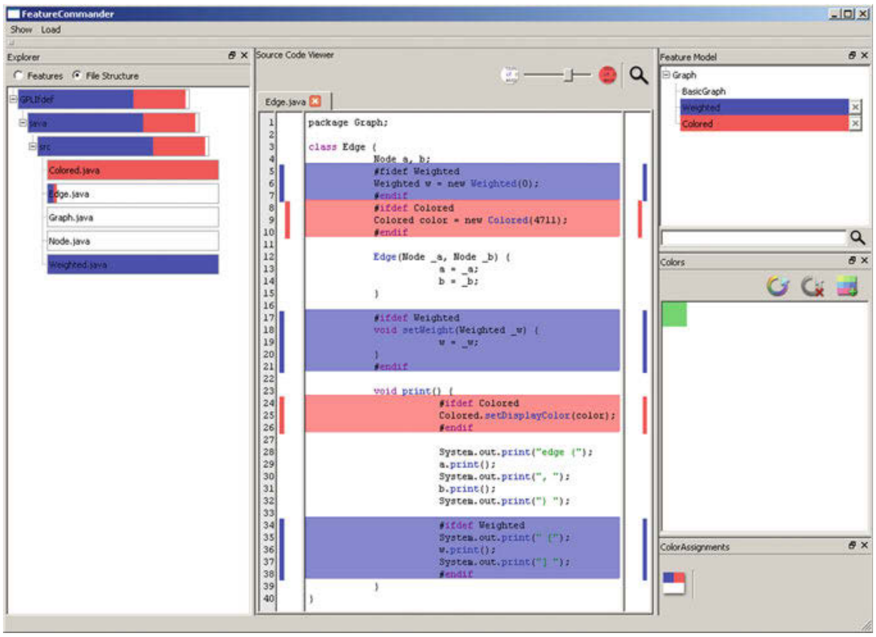
✓ BasicGraph	#FFFF0(
✓ Colored	#FF000(
✓ Weighted	#0000FI

Fig. 7.3 Graph implementation in CIDE; feature code is colored

features. Those code fragments will never be included in any product, because they cannot be selected by the user. Similarly, we can check whether a feature has any influence on the product derivation at all, otherwise we do not need to present the decision to the user. For many analyses discussed in Chap. 10, reasoning about the tracing information from the mapping is essential.

7.1.2 *Visualizing Tracing Information*

Tracing information can be *visualized* and presented to the user for comprehension and maintenance tasks. For example, with the tracing information from Fig. 7.2, the user can navigate between code fragments implementing a feature, even though the



```
package Graph;

class Edge {
        Node a, b;
        #fidef Weighted
        Weighted w = new Weighted(0);
        #endif
        #ifdef Colored
        Colored color = new Colored(4711);
        #endif

        Edge(Node _a, Node _b) {
                a = _a;
                b = _b;
        }

        #ifdef Weighted
        void setWeight(Weighted _w) {
                w = _w;
        }
        #endif

        void print() {
                #ifdef Colored
                Colored.setDisplayColor(color);
                #endif

                System.out.print("edge (");
                a.print();
                System.out.print(", ");
                b.print();
                System.out.print(") ");

                #ifdef Weighted
                System.out.print(" [");
                w.print();
                System.out.print("] ");
                #endif
        }
}
```

Fig. 7.4 Graph implementation in *FeatureCommander*; #ifdef-wrapped code is highlighted with background colors

mapping is not directly visible in the source code. In the tools CIDE and FeatureMapper, background colors (a distinct color per feature) are used to indicate feature code without visible #ifdef directives or other markers.

In Fig. 7.3, we show the annotated code of our graph example in CIDE, with the code of feature Weighted in red and the code of feature Colored in blue. For simplicity, the background color of the code of BasicGraph is not displayed (or rather, the color of BasicGraph is "clear").

Even when tracing information is visible in the source code (for example, in the form of #ifdef directives), tool support can still improve navigation beyond simple textual search, and it helps to overcome the code-obfuscation problem discussed in Sect. 5.3. For example, as illustrated in Fig. 7.4, the tool *FeatureCommander* highlights code fragments belonging to selected features with markers and background colors in the code editor. Markers in the file browser indicate which files contain code of the feature, markers in the margin of the text editor indicate where feature code is located inside a file, and background colors allow the programmer to visually find scattered and nested feature code quickly (even if the starting #ifdef is scrolled out of the current page).

Of course, tracing information can be visualized also in abstractions extracted from the implementation, such as dependency models, architectural models, and so forth. Several studies have indicated that visualizations of tracing information in product lines can improve program comprehension, navigation, and the overall performance in maintenance tasks (Feigenspan et al. 2012; Le et al. 2011). Feigenspan et al. (2012) discuss also limitations of colors and how to scale visualizations for product-lines with many features by using an on-demand mapping.

7.2 Views on Code

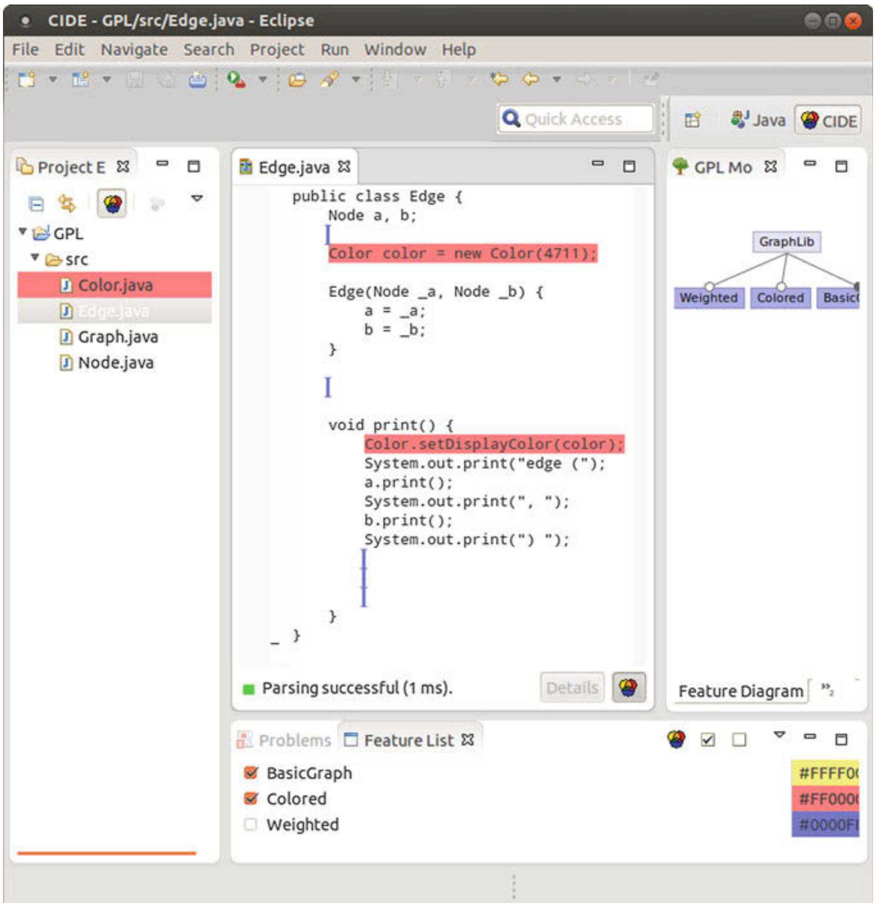
Given tracing links, a further opportunity to improve classic tool-based approaches is to emulate separation of concerns by providing views on source code (and noncode artifacts). One of the key motivations for separation of concerns (see Sect. 3.2.3, p. 55) is that developers can find all code belonging to a feature in one place, without being distracted by other code. A classic preprocessor-based implementation, for example, does not support this kind of locality, as the code of a feature is often scattered across the code base. By providing (editable) views on source code, advanced tool support can try to mitigate the problems and emulate separation of concerns.

Definition 7.2 In product-line development, a *view* on source code hides implementation details that are not relevant to a given feature selection. □

A typical view on a product-line implementation with many scattered features would show only the code of a particular feature or feature combination and hide the remaining code (similar to code folding). In Fig. 7.5, we illustrate the concept with a view in *CIDE* that hides feature Colored from the implementation of our graph example. The editor shows only code of the features BasicGraph and Weighted (features are annotated by the user and highlighted with background colors in the editor, as discussed in Sect. 7.1). Markers indicate hidden code, to make developers aware of hiding and to resolve ambiguities during editing. In addition to hiding code fragments inside files, also entire files can be hidden from the file navigator in a view. *CIDE* supports views on individual features and previews of generated products (that is, views on feature combinations).

Views allow the programmer to see all code belonging to a feature (possibly with some context information) in one place, even though it is physically scattered over many files. It is also possible to create a view on multiple features to inspect them all and their interactions. With views, developers can quickly explore feature code as if it was modularized. For this reason, Kästner has coined the term *virtual separation of concerns* (Kästner 2010), because separation is only emulated with views, but not actually enforced physically (see Sect. 7.4, p. 184).

In addition to views on individual features, views can also be used to give a preview on the source code that would be generated for a given feature selection on the fly (hiding all deselected features). This allows a developer to explore the structure and behavior of a product when multiple features interact more conveniently, without the



```
    public class Edge {
        Node a, b;
        Color color = new Color(4711);

        Edge(Node _a, Node _b) {
            a = _a;
            b = _b;
        }

        void print() {
            Color.setDisplayColor(color);
            System.out.print("edge (");
            a.print();
            System.out.print(", ");
            b.print();
            System.out.print(") ");

        }
  _ }
```

Parsing successful (1 ms).

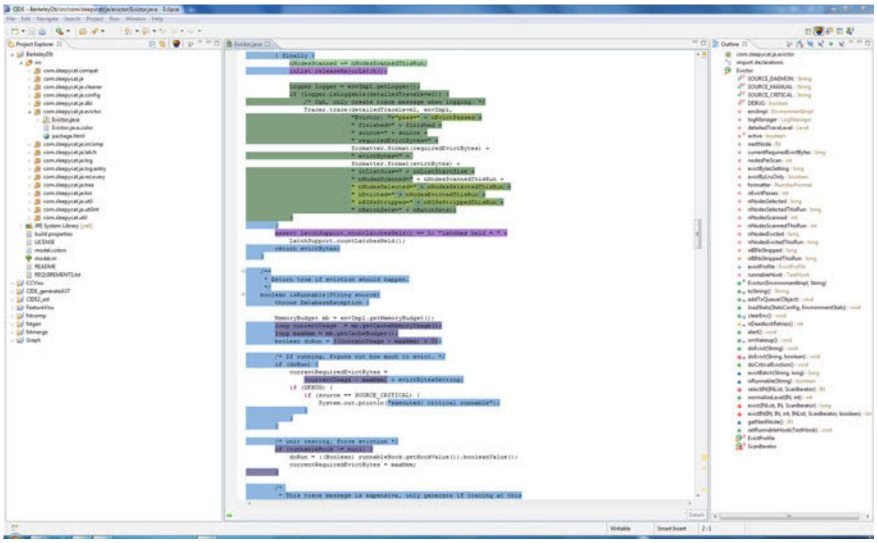
✓ BasicGraph	#FFFF0(
✓ Colored	#FF000(
☐ Weighted	#0000FI

Fig. 7.5 Graph library in CIDE; code of feature Colored is hidden; markers point to the hidden code

distracting code of unrelated features or having to trigger a lengthy build process to see the isolated code of multiple features.

The *Version Editor* takes this even a step further (Atkins 1998; Atkins et al. 2002). Given a preprocessor-like implementation, the version editor shows the code only of a single product (as if running the preprocessor with a feature selection). Changes to the code are propagated to the underlying product-line implementation (domain artifacts), such that they affect only the specified product (that is, with *#ifdef* directives around added and changed code to preserve the behavior of all other products).

Views have been explored in text and code editors (Atkins 1998; Singh et al. 2007; Kästner et al. 2008a,b; Batory et al. 2011), in model editors (Heidenreich et al. 2008a), integrated in version control systems (Chu-Carroll et al. 2003), and



```
        } finally {
            nNodesScanned += nNodesScannedThisRun;
            inList.releaseMajorLatch();

            Logger logger = envImpl.getLogger();
            if (logger.isLoggable(detailedTraceLevel)) {
                /* Opt, only create trace message when logging. */
                Tracer.trace(detailedTraceLevel, envImpl,
                             "Evictor: "+"pass=" + nEvictPasses +
                             " finished=" + finished +
                             " source=" + source +
                             " requiredEvictBytes=" +
                             formatter.format(requiredEvictBytes) +
                             " evictBytes=" +
                             formatter.format(evictBytes) +
                             " inListSize=" + inListStartSize +
                             " nNodesScanned=" + nNodesScannedThisRun +
                             " nNodesSelected=" + nNodesSelectedThisRun +
                             " nEvicted=" + nNodesEvictedThisRun +
                             " nBINsStripped=" + nBINsStrippedThisRun +
                             " nBatchSets=" + nBatchSets);
            }
        }
        assert LatchSupport.countLatchesHeld() == 0: "latches held = " +
            LatchSupport.countLatchesHeld();
        return evictBytes;
    }

    /**
     * Return true if eviction should happen.
     */
    boolean isRunnable(String source)
        throws DatabaseException {

        MemoryBudget mb = envImpl.getMemoryBudget();
        long currentUsage  = mb.getCacheMemoryUsage();
        long maxMem = mb.getCacheBudget();
        boolean doRun = ((currentUsage - maxMem) > 0);

        /* If running, figure out how much to evict. */
        if (doRun) {
            currentRequiredEvictBytes =
                (currentUsage - maxMem) + evictBytesSetting;
            if (DEBUG) {
                if (source == SOURCE_CRITICAL) {
                    System.out.println("executed: critical runnable");
                }
            }
        }

        /* unit testing, force eviction */
        if (runnableHook != null) {
            doRun = ((Boolean) runnableHook.getHookValue()).booleanValue();
            currentRequiredEvictBytes = maxMem;
        }

        /*
         * This trace message is expensive, only generate if tracing at this
```

Fig. 7.6 Fine-grained annotations in Berkeley DB with CIDE

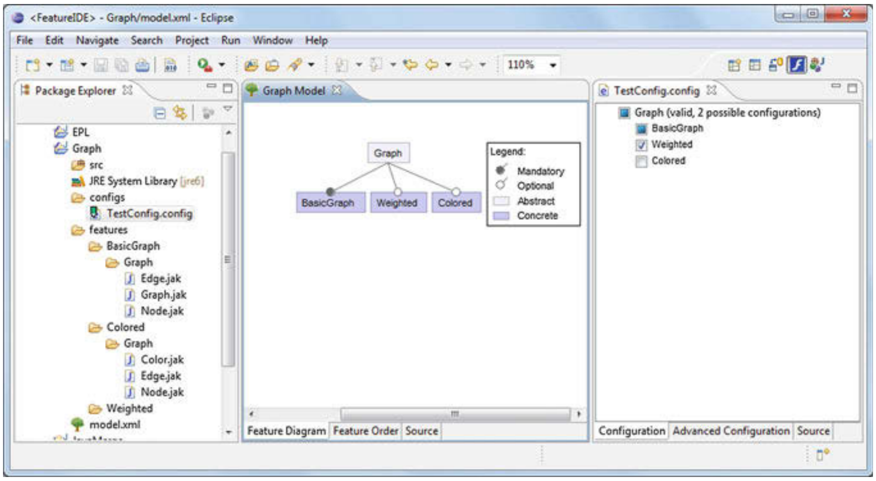
even directly in a file system (Hofer et al. 2011). To approach real modularity with information hiding, Janzen and De Volder (2004) even attempt to rewrite the source code physically on demand, called effective views, and Ribeiro et al. (2010) compute interfaces for views on the fly.

When many fine-grained features interact, such as in Berkeley DB (Fig. 7.6), views can be a tremendous help (Kästner 2010). Atkins et al. (2002) found that views in the *Version Editor* improve developer productivity in variation-rich code by 40 %, compared to standard editors without views.

7.3 Integrated Product Derivation

Many modern tools integrate all steps of the process of product-line development (see Sect. 2.2, p. 19). They do not necessarily provide a novel variability-implementation mechanism, but rather integrate existing approaches (parameters, frameworks, build systems, version control, feature-oriented programming, tool-driven tracing, and so forth) and combine these with feature models as central artifacts describing and connecting features in the system.

A tight integration of all development phases can pay off, especially, during application engineering and product derivation, when we want to derive a product for specific customers and their requirements. Tools can provide support in two ways: *checking validity* of feature selections and *automation of product derivation*. Without integrating feature models, developers were often left alone with figuring out how



Graph (valid, 2 possible configurations)
- ■ BasicGraph
- ✓ Weighted
- ☐ Colored

Fig. 7.7 Support for product derivation in FeatureIDE

to configure parameters and how they interact, or how to assemble components, or which plug-ins are compatible.

Instead of an arbitrary list of configuration options, feature models provide guidance to users configuring a product. The hierarchical structure of feature models gives structure to the configuration space, and feature descriptions (see Sect. 2.3.1, p. 27) explain the functionality and rationale for each feature. Dependencies between features are documented explicitly and can be automatically enforced, for example, by automatically propagating feature selections to dependant features or by deactivating mutually exclusive features.

Advanced analyses have been proposed to guide the process of product derivation (Should this feature selected or an alternative feature? What features are selectable at a certain point during the configuration process?) and to predict the effect of a feature selection on nonfunctional properties (see Chap. 10). All these analyses can be used to guide stakeholders in making efficient decisions during product derivation. Finally, given a feature selection, an integrated tool often can automate the actual generation of the product, for example, by running build systems, generating configuration files, or controlling the composition process of plug-ins, feature modules, and aspects, as described in the previous chapters.

As example, consider the screenshot from the tool *FeatureIDE* in Fig. 7.7. In the middle, we see the feature model of our graph example. On the left, we see the project overview (implemented with feature-oriented programming). On the right, we see a feature-selection dialog that can be used interactively to derive a product by selecting features. Internally, *FeatureIDE* translates both the feature model and the feature selection to a propositional formula, and it uses a satisfiability solver to check the validity of a feature selection, as we will explain in detail in Chap. 10. The user gets immediate feedback which features are still available, given that some are already selected and deselected, and how many products are still possible to derive. The corresponding product is automatically generated in the background. Similar functionality has found its way into commercial product-line tools and also the Eclipse update manager, which checks dependencies between plug-ins before installing features.

7.4 Discussion: Virtual Separation of Concerns

We have discussed various possible improvements in product-line tooling. We call the combination virtual separation of concerns.

Definition 7.3 *Virtual separation of concerns* is a tool-based approach to product-line development. Based on a combination of tracing information, visualization facilities, source-code views, and the integration of feature models, code that belongs to individual features can be displayed, edited, and understood in separation. □

We see virtual separation of concerns as an improvement over the limitations of classic tool-driven approaches (see Chap. 5) and as a full-fledged alternative to advanced, language-based approaches (see Chap. 6). The key idea is to provide tool support to facilitate a separation of concerns without relying on language mechanisms such as class refinements or advice, but to achieve similar benefits.

Tracing information is typically used to customize code at compile-time (see *binding times* in Sect. 3.1.1, p. 48; though load-time and run-time binding is conceptually possible as well by injecting variability mechanisms into the source code before compilation). Feature traceability is a key goal and exploited in many forms for navigation, visualization, and analysis (see *feature traceability* in Sect. 3.2.2, p. 54). Explicit, disciplined feature traceability actually facilitates writing tools, for example, for refactoring (see Chap. 8) and analysis (see Chap. 10).

Despite tool improvements, virtual separation of concerns preserves the general mechanisms and feel of preprocessor-based implementations. Tools use annotation-based mechanisms (see *annotation versus composition* in Sect. 3.1.3, p. 50) and operate at arbitrary levels of granularity (see *granularity* in Sect. 3.2.5, p. 59). Also similar to preprocessors, little upfront preplanning is required and code is rather changed invasively (see *preplanning* in Sect. 3.2.1, p. 53), which aligns well with *extractive product line adoption* (see *adoption paths* in Sect. 2.4, p. 39). The mechanisms remain lightweight and easy to learn and adopt. Although based on structures of the source code, the discussed tool improvements can be typically applied uniformly to code and noncode artifacts (see *principle of uniformity* in Sect. 3.2.6, p. 60).

The systematic integration of the feature model in tools combats the scattering of configuration knowledge, and visualizations and views attempt to limit code obfuscation, for which preprocessors are often criticized (see Sect. 5.3.6, p. 120). The structured nature with disciplined annotations helps to prevent basic syntax errors and enables more sophisticated analyses, as discussed in Chap. 10.

Approaches for virtual separation of concerns do not physically separate code of different features. Instead, the code is intermixed in a common code base. Unlike classic preprocessors though, views allow programmers to virtually separate code of different features, which facilitates separation of concerns (see Sect. 3.2.3, p. 55). For example, one can view the code only of a certain feature or feature combination, whereas the remaining code is hidden. Additionally, a developer can get a view on the combined code of multiple feature (thus seeing the interacting code fragments in place), which is not easily possible in language-based approaches. Whereas virtual separation of concerns assists the programmer in understanding the system in terms of the features it provides, it does not support information hiding with interfaces and separate compilation though (see Sect. 3.2.4, p. 57).

Compared to advanced language-based approaches (Chap. 6), virtual separation of concerns provides a simpler underlying mechanism based on annotations. It remains limited regarding information hiding, but excels, especially, at fine-grained extensions, where composition-based approaches are limited.

Summary virtual separation of concerns
Strong points:

- Simple programming model: annotate and conditionally remove.
- Structured tracing of features (see Sect. 3.2.2, p. 54).
- Compile-time customization of source code; no boilerplate code (see Sect. 3.1.1, p. 48).
- Flexible, and support of arbitrary granularity (see Sect. 3.2.5, p. 59).
- Little preplanning required (see Sect. 3.2.1, p. 53).
- Virtual separation of concerns, despite physical scattering and tangling of feature code (see Sect. 3.2.3, p. 55),
- Lightweight mechanism for extractive product-line adoption.
- Uniform application to source code and noncode artifacts (see Sect. 3.2.6, p. 60).
- Easy to use and analyze.

Weak points:

- No support for information hiding (see Sect. 3.2.4, p. 57).
- Mostly academic tools so far, little experience in practice.
- Many fined-grained and interleaved annotation hinder program comprehension.

7.5 Tooling

Ideas in this chapter have been explored individually and in combination in various tools, mostly academic prototypes.

There are several tools which manage a mapping between features and code fragments inside the tool infrastructure. Tools that support basic feature tracing (but no further advanced mechanisms) are, for example, *FEAT* and *ConcernMapper*. *FEAT*[1] is an Eclipse plug-in that supports feature modeling and tracing. *ConcernMapper*[2] is more general in that all kinds of concerns are supported; there is no support for full-fledged feature models. The tools *CIDE*,[3] *FeatureMapper*,[4] and *fmp2rsm*[5] go further and use the mapping also for generating products given a feature selection, such functioning as a form of preprocessor.

Various visualizations of the mapping between features and their implementation artifacts have been implemented in *FeatureCommander*,[6] *FeatureMapper*, *CIDE*, and *View Infinity*.[7]

Views have been explored in several tools. Unfortunately, to the best of our knowledge, only *CIDE* and *FeatureMapper* are available for experimentation.

FeatureIDE[8] is a tool suite for product-line development that supports feature modeling and tracing, but also feature-guided product derivation. *Gears*[9] and *pure::variants*[10] are two commercial tools that support feature-guided product derivation in different ways incorporating different technologies.

7.6 Further Reading

There is no standard literature on tool-based approaches to feature-oriented product lines. Of course, product-line tools are covered by the standard books on software product lines (Clements and Northrop 2001; Pohl et al. 2005). Greenfield and Short introduce the related concept of a *software factory*—a tool infrastructure for the generative development of software systems based on models, patterns, and frameworks. Software factories can be used to develop software product lines using a tool-based approach, and the tools we have discussed in this chapter can be viewed as software factories.

[1] http://www.cs.mcgill.ca/~swevo/feat/

[2] http://www.cs.mcgill.ca/~martin/cm/

[3] http://fosd.net/cide/

[4] http://featuremapper.org/

[5] http://gsd.uwaterloo.ca/fmp2rsm

[6] http://fosd.net/fc/

[7] http://fosd.net/vi/

[8] http://fosd.net/FeatureIDE/

[9] http://www.biglever.com/solution/product.html

[10] http://www.pure-systems.com/pure_variants.html

Kästner (2010) provides a comprehensive introduction to virtual separation of concerns in his dissertation; a corresponding journal article summarizes the main ideas (Kästner and Apel 2009). There is a whole set of proposals that are related to or instances of virtual separation of concerns, mainly published in scientific papers (Atkins 1998; Singh et al. 2007; Batory et al. 2011; Heidenreich et al. 2008a; Chu-Carroll et al. 2003; Hofer et al. 2011; Janzen and De Volder 2004; Ribeiro et al. 2010).

Exercises

7.1 Reimplement the chat system (Exercise 4.1, page 96) with CIDE or a similar tool (see Appendix A).

(a) How does CIDE manage traceability links and enforces consistency?
(b) Explore views on the source code (views on features and views on products). Is it feasible to understand and maintain (possibly scattered) implementations of the features in isolation? Explore whether views help with the following two maintenance tasks:

- Change the format in which colors are exchanged between client and server (only when feature Color is active).
- Add a message to the history whenever a user successfully logs in (only when features History and Authentication are both activated).

(c) Compare your experience with the preprocessor implementation (Exercise 5.4, page 125). Is CIDE's restriction to disciplined annotations or the confinement to a specific tool environment limiting? Does tool-based traceability add any practical benefits?
(d) Compare the implementation in CIDE with previous implementations using feature-oriented and aspect-oriented programming (Exercises 6.2 and 6.4, page 172 and 173), especially with regard to feature traceability, separation of concerns, information hiding, and program comprehension.

7.2 Discuss the potential benefits and drawbacks of virtual separation of concerns. To what degree can tool support to replace physical separation of concerns? Does virtual separation of concerns discourage preplanning and designing for change, and thus introduce bad habits for developers? Does virtual separation of concerns require commitment to a specific tool (vendor locking)? Can it scale for large projects with many developers? Can or should it be integrated with other variability-implementation approaches?

7.3 Brainstorm possible uses of traceability information, maintained in a tool infrastructure. What benefits or problems would arise from tracking not only variable features of a product line used during product derivation, but also concerns and design decisions that may be the same in all products? Do you know any existing tools that could be used to trace features or that already exploit feature traceability? Can you envision any direct use of the traceability information from Exercise 7.1?

7.4 Draw a diagram consisting of all phases of the product-line-development process (see Fig. 2.1). Enhance this diagram with connections between the phases that could be exploited with tool support. A possible example could be that the domain-implementation phase can use information about available features in the domain-analysis phase to check the correctness of annotations. Explore further potential for integration and tool support.

7.5 Reconsider the scenarios of Exercise 2.9 (page 44). Under which circumstances would you recommend adopting any of the additional tool support discussed in this section?

7.6 Complete the comparison of Exercise 4.11 (page 97) by adding virtual separation of concerns.

Exercises Spanning Entire Part II

7.7 After the discussion of many different variability-implementation approaches, the ultimate question is which approach should be used for a practical product-line project. Unfortunately, there is no single solution that serves all needs. Provide a technical consulting service for the developers of the following scenarios and recommend an implementation approach or a combination of implementation approaches and a possible migration strategy where applicable. Asses the involved risk. Explain why you rule out specific adoption strategies and implementation mechanisms or why you prefer others in each scenario. Use the quality criteria from Chap. 3 for your arguments where suitable. The scenarios are all intentionally short and underspecified; describe which additional context knowledge you would need for a decision or which additional knowledge would change your recommendation.

(a) A small company selling universal remote controls has had a long history with only two separately developed products. More recently resellers have requested specialized editions that required small modifications to the firmware (mostly different startup logos, different pre-configurations of channels, and different algorithms for controlling battery loading). The company focuses mostly on hardware; the firmware has been developed by a single developer in a C dialect specialized to produce small binaries for micro-controllers. To accommodate the modification requests of resellers the developer has used branches in a version control system. Lately, the developer maintains 15 branches and complains about increasing maintenance effort.

(b) A small startup specializes in developing simple games for smartphones with a marketing message. The start-up plans to quickly build games for big corporate customers. Quick production at reasonable quality is the key goal. The start-up has a small team of developers and designers who work on new projects that come up with a fresh new concept for every game. Technically, the games are all based on HTML5 and JavaScript on the smartphone, and Ruby on the backend.

(c) A development group of a large software company focused on graphics and multimedia software plans to release a new easy-to-use video-editing software. A low-end version of the software should be release for free, for which consumers can buy advanced functionality (editing tools, filters, formats, and so forth) individually at low prices. To keep revenue, the company plans to frequently release new functionality. Development has not started yet, but the company is willing to provide a sizable budget and an experienced team. Functionality from existing high-end professional products, written in C++, may be reused, while making sure that the new low-end product does not compete with the professional products.

(d) A group of enthusiastic developers builds hardware and writes software for a quadrotor helicopter in their free time and shares the results as open-source hardware and open-source software. They experiment with various hardware alternatives and extensions, including various sensors, motors, and radio equipment. Their software should be flexible for extensions and experimentation and adjustable to different hardware parts used by others rebuilding the helicopter. Code quality and extensibility is important, but also binary size, performance, and energy consumption should not be neglected entirely.

(e) A team of open-source developers wants to implement a nerdy and cool window manager for Linux. The window manager will make heavy use of graphical effects and can be controlled over various input methods (keyboard, touch, voice, and so forth). The system should be extremely configurable so that no two installations look alike and that the behavior can be tailored precisely to the personal preferences. The initial implementation is based on Python and C++.

(f) A group of computer science graduate students from different institutions wants to build an infrastructure for testing and benchmarking communication protocols as part of their research. The infrastructure must be flexible and configurable to connect to different protocols over different communication links under different loads.

(g) The three core developers of a new and hyped NoSQL database system for distributed and high-load web sites want to experiment with introducing distributed transactions. Their system, written in Java, was especially fast since it had no transaction support. Now the developers want to experiment with different transaction strategies, to determine which may be suited for their needs. Potentially, they may want to offer different strategies (or none) to their customers. Transaction systems are notoriously crosscutting and the developers want to ensure that development of and experimentation with transaction mechanisms does not interfere with their core product (without transactions) that is continuously extended and maintained in the meanwhile.

(h) A development team in a large software company explores extensions of a middleware platform that was previously developed in Java by open-source developers paid by several other companies on the open-source platform *Github*. Several forks of the project already exist, but, except for one, the forks are hardly maintained. The developers fork the project again and plan to add multiple features that are important in their proprietary company context and they plan to experiment with alternative implementations. In the long run, they consider contributing all or some of their changes back to the original project, if the original developers are interested.

Part III
Advanced Topics

Chapter 8
Refactoring of Software Product Lines

After reading the chapter, you should be able to

- characterize the basics of refactorings in object-oriented source code,
- discuss characteristics of refactorings in software product lines,
- describe and find variability-specific code smells,
- perform variability-related refactorings, and
- select and perform suitable refactorings for product-line adoption and explain their chances and limitations.

Refactoring is an important activity in software development, maintenance, and evolution. Refactorings are changing the structure of software, without changing its behavior. Refactorings are typically performed to improve the code structure, for example, to make code more readable or to prepare it for an extension. Typically, refactorings are performed as a reaction to a code smell, a perceived problem in the source code. Refactorings are a core ingredient for agile development practices, but have been generally embraced by developers in practice. Many refactorings can be performed automatically by modern development environments.

In the context of product lines, the notion of refactorings is interesting, because one develops not only a single product, but a set of domain artifacts that may be used to generate a whole family of products. For example, we could expect that a refactoring may not change the behavior of any derivable product, but also more relaxed notions are useful, as we will discuss. Also, software product lines give rise to a new group of code smells that arise from the use of features and variable implementations. Finally, refactorings can be used in the adoption phase of product-line technology, for example, by extracting and rewriting reusable parts of legacy systems.

Refactoring in software product lines is still a young topic, which needs further exploration. In this chapter, we will give an overview of different notions of refactorings and possible avenues for research and practice of refactorings in feature-oriented software product lines.

DOI: 10.1007/978-3-642-37521-7_8, © Springer-Verlag Berlin Heidelberg 2013

8.1 Refactoring in General

Before we get to the specifics of refactoring product lines, let us lay a foundation by introducing traditional object-oriented refactorings. Refactorings are typically traced to the dissertation of Opdyke (1992) and have been popularized by the seminal book of Fowler (1999). Fowler defines refactoring as a process as follows:

Definition 8.1. *Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. (Fowler 1999) □

A refactoring is typically performed in response to a perceived problem of the structure of existing code, called "code smell." The term "code smell" is used in an informal fashion for an indicator of a potential problem, such as very long methods. A very long method can be a sign of poor program structure that could be worth improving, for example, by splitting the method into multiple smaller methods to simplify understanding and maintenance. However, a very long method by itself is not a bug and not always worth changing (depending on the context), hence the rather vague term "smell." (Preferably, code smells are defined in a measurable form to enable tool support that points developers to possible locations.)

Definition 8.2. A *code smell* is a perceivable property of source code that is an indicator of an undesired code property. □

Fowler (1999) characterize code smells as "indications that there is trouble that can be solved by refactoring." They collected a long list of code smells in object-oriented programs, including

- *Duplicate code*: Duplicate code is an indicator for missed abstraction possibilities. Instead of repeating code, one could consider to abstract it into a shared function or class.
- *Large class*: A large class with many fields and methods may become difficult to understand and may no longer focus on a single abstraction. Developers could consider splitting the class into smaller classes with clear responsibilities.
- *Shotgun surgery*: When maintenance or evolution tasks often require performing crosscutting changes throughout many classes, this could be an indicator for inadequate separation of concerns (see Sect. 3.2.3, p. 55 and our discussions of modularity throughout Part II). Instead of scattering code and changes, developers could consider separating concerns more explicitly.

To eliminate code smells, developers typically know a set of common strategies to change code without changing its behavior, such as splitting methods, extracting shared code, moving code fragments, and many more. These common behavior-preserving transformations are called refactoring steps (note, the term refactoring is overloaded to denote both the overall process of improving the structure of code as well as refactoring steps, the individual code transformations).

Definition 8.3. A *refactoring* or *refactoring step* is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

(Fowler 1999) □

To facilitate communication and sharing among developers, refactorings have been collected in catalogs and described using a uniform structure (typically containing name, motivation and potential gain, addressed code smells, mechanics for the actual transformation, preconditions for its applicability, and examples)—similar to collections of design patterns (see Sect. 4.2, p. 69). Refactoring steps are usually described in the form of recipes to be performed manually. However, many refactoring steps have been implemented in modern development environments and can be performed automatically (the development environment then checks preconditions and performs the rewrites).

Examples of typical refactorings, collected by Fowler (1999), are:

- *Renaming method*: Change the name of a method and of all corresponding method invocations. This refactoring can address a code smell of inadequately named methods.
- *Move method*: Move a method to a different class, while updating all the invocations of the method. This refactoring addresses, among others, the code smells shotgun surgery and large class.
- *Extract method*: Move a sequence statements to a new method introduced for this purpose and call this method from the previous location. This refactoring addresses, among others, the long-method code smell and, as a preparatory step, the duplicate-code code smell.

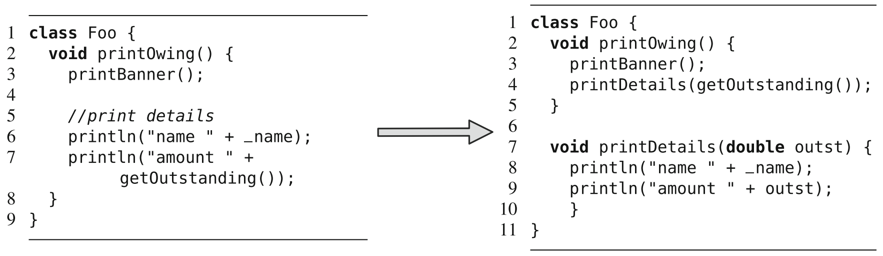
Much like design patterns, refactorings have been explored in many languages and domains, including refactorings for object-oriented languages Fowler (1999), aspect-oriented languages Monteiro and Fernandes (2005), and graphical models Sunyé et al. (2001); Arendt et al. (2010). In the remainder of the chapter, we will concentrate on refactorings for product lines.

On the Correctness of Refactorings

Refactorings are, by definition, behavior preserving. However, they are often described as technical recipes, to be executed manually. Many refactorings have intricate preconditions and require upfront analysis of the entire code base (for example, finding all calls to a function). In fact, performing refactorings manually can easily introduce bugs, and even automated refactorings implemented by development environments have been shown to be frequently incorrect in corner cases. For instance, the seemingly innocent extract-method refactoring has quite complex implications when the extracted code contains assignments to local variables (Murphy-Hill and Black, 2008).

Refactorings are rarely formalized, as are the languages in which they are performed. Although there is research on formally proving behavioral equivalence of the code before and after applying a refactoring (Estler et al., 2007; Sultana and Thompson, 2008), most refactorings are not formally verified. A typical strategy to ensure correctness is to define very small refactoring steps that are easy to check for correctness and then build larger refactoring steps by combining smaller ones (Kniesel and Koch, 2004; Cole and Borba, 2005). The standard practice relies on testing of the refactoring engine (if automated) and regression testing of the refactored code (especially, if refactored manually). With this caveat in mind, we return to the happy land where refactorings are, by definition, "correct."

Example 8.1 In Fig. 8.1, we illustrate the *Extract Method* refactoring. A programmer decides to extract some statements of a method (Lines 6–7 on the left) into a separate method printDetails. Extracting the statements and moving them to a new method should not change the observable behavior of the program. □



```
1 class Foo {
2   void printOwing() {
3     printBanner();
4
5     //print details
6     println("name " + _name);
7     println("amount " +
           getOutstanding());
8   }
9 }
```

```
 1 class Foo {
 2   void printOwing() {
 3     printBanner();
 4     printDetails(getOutstanding());
 5   }
 6
 7   void printDetails(double outst) {
 8     println("name " + _name);
 9     println("amount " + outst);
10     }
11 }
```

Fig. 8.1 Example of a typical refactoring: *Extract Method*

8.2 Refactoring in Software Product Lines

How does refactoring interplay with software product lines? What does it mean to change the structure but preserve behavior in a product line, in domain artifacts, in feature models, or in generated code? Are there code smells specific to product-line implementations? Does the notion of refactoring in product lines differ between different implementation mechanisms?

First of all, of course, traditional object-oriented refactorings can also be applied to implementation artifacts in product lines, to eliminate well-known code smells such as long methods, large classes, or duplicate code. For example, we can use an extract-method refactoring also in parameter-based implementations (see Sect. 4.1, p. 66), inside plug-ins of a framework (see Sect. 4.3, p. 79), in a method surrounded by conditional-compilation statements (see Sect. 5.3, p. 110), in the body of a piece of advice of an aspect (see Sect. 6.2, p. 141), and so forth. Especially, for classic language-based variability-mechanisms, existing tool support in development environments can be used.

However, in our context, the interplay between code artifacts and variability is more interesting. Let us start with a number of smells that occur in software product line and subsequently discuss possible refactorings.

8.2.1 Variability Smells in Software Product Lines

Also, in product lines, there are many smells which indicate potentially inadequate feature modeling or inadequate implementations. To emphasize the difference focus on variability, let us name them *variability smells*. As with traditional smells, a variability smell is an indicator of a potential design problem, but is not a problem by itself.

Definition 8.4. A *variability smell* is a perceivable property of a product line that is an indicator of an undesired code property. It may be related to all kinds of artifacts in a product line, including feature models, domain artifacts, feature selections, and derived products. □

Let us collect a list of examples for variability smells:

- *Obscure feature model*: During application engineering, stakeholders have problems mapping their requirements to the features of the product line. This can indicate suboptimal naming and documentation of features, an unsuitable structure of the feature model where features are located in unsuspected branches, or overly complex and hard to understand cross-tree dependencies in the feature model. Developers may rename features or restructure the feature model to ease understanding.

- *Unused variability*: A feature is modeled and implemented as optional feature, yet it is selected in all products currently derived. Although we might explicitly decide to keep the feature optional for future customers, unused variability can be an indicator of variability that is not actually needed, so developers might decide to remove the induced overhead.
- *Unused feature*: The inverse of unused variability is a feature that is optional, but not selected in any product currently derived. Again, we might decide to keep the feature for future customers, but unused features can also be an indicator that encourage removing unnecessary code.
- *Dead feature*: A feature modeled as optional feature in the feature model, but actually not selectable due to constraints (for example, a mutually exclusion constraint with a mandatory feature) is called a dead feature. A dead feature might be an indicator of an incorrect feature model or a feature that is no longer needed.
- *Dead feature code*: A code fragment is dead if it can never be included in the derivation of any derivable valid product, for example, because it requires the selection of two mutually exclusive features. Dead feature code can easily occur in annotative implementation strategies, from runtime parameters, to conditional compilation, and to build systems. Dead feature code is a strong indicator for a problem in the feature model, in the implementation, or the mapping between features and code fragments (see also Sect. 10.2).
- *Fat products*: Despite tailoring to user needs, the derived products are still unnecessarily large and contain unneeded functionality. This can be an indicator for insufficient variability. Developers may decide to identify additional features and make the corresponding code fragments optional.
- *Dependency clusters*: Intricate implementation dependencies between features may overly restrict variability during feature selection. Developers may take this as indicator to restructure the implementation and decouple feature implementations (see also the optional-feature problem in Chap. 9) or alternatively merge those features into a single feature.
- *Duplicate code in alternative features*: Two alternative features replicate code in their implementation. Developers may take this as an indicator to extract the common code into a shared implementation unit (plug-in, feature module, aspect, and so forth) that is imported by both features. It may also be an indicator of unsuitable granularity provided by the variability mechanisms (for example, file-level granularity of build systems, Sect. 5.2, p. 105) and encourage selecting a different implementation strategy.
- *#ifdef hell or many extension points*: Depending on the implementation strategy, the implementation of a feature may be scattered across the code base (see Sects. 4.1.1 and 5.3.6, p. 66 and 120) or require many scattered extension points (see Sect. 4.3.5, p. 86). Like the code smell shotgun-surgery, this can be an indicator for suboptimal separation of concerns. Developers may want to invest into a redesign or consider changing the implementation strategy.

- *Runtime overhead*: Many runtime checks for features or dynamic dispatch may cause overproportional performance penalties during execution, especially in small, embedded systems (see Sects. 4.1.1 and 4.3.5, p. 66 and 86). Developers can take this as indicator to consider changing load-time bindings into compile-time bindings, for example, changing from *if* statements to *#ifdef* directives or feature modules.
- *Binary chaos*: Compile-time binding requires regenerating and recompiling the project for every new feature selection. With many customers or many changes, the compilation overhead or the storage space required for the many binaries may be excessive. Developers can take this as an indicator to change compile-time bindings into runtime bindings that can be configured with configuration files or command-line parameters.
- *Traceability mess*: Developers struggling to find the implementation of a feature, especially in parameter-based and build-system-based implementations (see Sects. 4.1.1 and 5.2.4, p. 66 and 108), may be seen as an indicator to use additional traceability tools (see Sect. 7.1, p. 175) or to use a different implementation strategy.
- *Language and variability-mechanism overload*: In a project with artifacts in different code and noncode languages, developers struggle with a mix of many different variability-implementation mechanisms (for example, parameters combined with build systems and conditional compilation and aspects) and extensions for many different languages (for example, AspectC, AspectUML, and AspectJ). Developers may take this as an indicator to rewrite variability implementations with fewer and more uniform implementation mechanisms.
- *Limited extensibility*: After significant preplanning effort and commitment to noninvasive extensions, developers struggle with finding suitable extension points. Developers might take this as an indicator to redesign extension points or to adopt a more invasive implementation strategy.

Research on variability smells is still in its early steps. A next step will be to categorize these example smells into proper categories.

The list of feature code smells is meant to give an overview of the wide variety of possible code smells in product-line implementations, but the field is too young and in a state of flux to propose a comprehensive catalog. Note that these smells address all facets of product-line implementations, from feature models and feature selections to feature implementations. As discussed in Part II, developers need to decide between many trade-offs regarding, for example, binding times, preplanning effort, granularity, and uniformity. Some smells are linked to specific trade-offs, such as deciding to use an invasive and crosscutting implementation strategy such as conditional compilation.

For many code smells, we can identify corresponding changes to the product line to eliminate those smells, such as adding, removing, and merging features, rewriting variability implementations, and changing binding times. For many of these changes we can describe instructions for transformations that preserve the key behaviors in the product line.

8.2.2 Defining Product-Line Refactorings

Refactorings in product lines aim at improving the structure while generally preserving observable behavior. Previously, we raised the question of what behavior preserving means for a product line. For example, is adding a feature to a product line while leaving existing products unchanged preserving behavior; is changing binding times preserving behavior?

First, we need to decide for which products we want to preserve behavior:

- Preserving the set of possible valid products and the behavior of all these products is the most restrictive definition of product-line refactoring, which we name *variability-preserving refactoring*. A variability-preserving refactoring may not remove or add products to a product line and may not change the observable behavior of any potential or actually delivered product. This strict notion of behavior preserving is useful if we want to improve the structure of the product line, but make sure that we do not break any existing behavior or accidentally introduce additional behavior.
- Preserving the behavior of all valid products (but not necessarily preserving the exact set of valid products) is a less restrictive definition of product-line refactoring, which we call *variability-enhancing refactoring*. It allows us adding new variations in order to be able to derive additional products (for example, adding features, removing constraints), but not removing or affecting the behavior of products that were previously possible. This slightly relaxed notion of behavior preserving is useful if we want ensure backward compatibility when evolving the entire product line to broaden its scope.
- Preserving the behavior of products currently derived for customers is the least restrictive definition of product-line refactoring, which we name *product-preserving refactoring*. These refactorings may remove or add features, as long as they do not change the behavior of a given *subset* of valid products. This relaxed notion of behavior preserving is useful if we care only about a specific subset of products—potentially products currently delivered to customers—but want flexibility to change the remainder of the product line.

Second, we have a certain flexibility regarding which transformations we consider as refactorings:

- *Renaming features*: Technically, renaming a feature in the feature model does not change the total number of possible products, but does change the set of possible feature selections (see Sects. 2.3 and 10.1.7, p. 26 and 252), hence, we can argue that renaming a feature is not a refactoring that preserves the variability of a product line. However, considering a more holistic view of product-line development, we can argue that renaming a feature would also rename the feature in all feature selections and yield an equivalent set of possible feature selections.
- *Changing binding times*: Technically, changing the binding time of a feature from compile-time binding to load-time binding, or vice versa, changes the possible behavior of the compiled product. However, we argue that, conceptually, the

product line preserves variability and runtime behavior of the product in equivalent feature selections.

In summary, we adopt the following definitions:

Definition 8.5. A *product-line refactoring* is a refactoring step specific to feature-oriented product-line development that may affect the feature model and the domain artifacts of a software product line.

A *variability-preserving refactoring* is a product-line refactoring that does not change the set of valid products and corresponding feature selections (modulo renaming) and preserves the observable behavior of all valid products (modulo binding time changes).

A *variability-enhancing refactoring* is a product-line refactoring that does not remove products from the set of valid products and preserves the observable behavior of all previously valid products. A variability-enhancing refactoring may introduce additional valid products.

A *product-preserving refactoring* is a product-line refactoring that does not remove products from a given set of products and preserves the observable behavior of all those products. A product-preserving refactoring may add and remove products outside the given set. □

8.2.3 Examples of Product-Line Refactorings

With Definition 8.5, we can now come back to refactorings that address variability smells, as introduced in Sect. 8.2.1. Again, we do not strive for completeness but for breadth to illustrate the large design spectrum of possible refactorings. Designing a comprehensive catalog of product-line refactorings and formally ensuring correct behavior and variability preservation of these refactorings is an interesting research direction (see also further readings below).

- *Move feature*: Rearranges a feature in the feature model to address the smell obscure feature model (for example, by moving a feature closer to semantically related features in the feature model). When all valid feature selections are preserved, this is a variability-preserving refactoring that does not even affect the implementation.
- *Rename feature*: Variability-preserving refactoring that renames a feature consistently in the feature model, in existing feature selections, and in the implementation and mapping. For instance, in preprocessor-based implementations this may require renaming flags in all *#ifdef* directives. This refactoring can address variability smells, such as obscure feature models.

- *Delete feature*: Product-preserving refactoring that removes an unnecessary feature from the feature model and removes all corresponding implementation artifacts. If the feature or feature implementation was dead, the removal is even a variability-preserving refactoring. This refactoring can address smells such as unused feature, dead feature, and dead feature code.
- *Merge features*: Product-preserving refactoring that merges the implementation of two features to address the smells unused variability and dependency cluster. A special case is merging a feature with the base code, that is, removing variability. Depending on the implementation mechanism, developers may merge two *#ifdef* flags or combine two plug-ins, feature modules, or aspects into a single one, as illustrated in Example 8.2.
- *Extract feature*: Variability-enhancing refactoring that adds a new feature to the feature model and maps it to existing implementation artifacts. The refactoring preserves existing behavior (when the new feature is selected), but allows users to deactivate functionality, which can address the fat-product smell. Depending on the implementation approach, extracting a feature may require introducing new parameters and *if* statements, to inject *#ifdef* directives, or to move code to separate implementation units (similar to object-oriented extract-method or move-method refactorings). We discuss the extract-feature refactoring in the context of an extractive adoption strategy in Sect. 8.3, including some examples.
- *Extract shared code*: Variability-preserving refactoring that moves code replicated between two features into a new artifact that is included by both features, to address the smell duplicate code. Depending on the implementation strategy, developers may annotate shared code properly (for example, with a disjunction of both features) or move the code into a separate plug-in, feature module, or aspect that is automatically included with either feature.
- *Change binding-time*: Variability-preserving refactoring to change the existing variability-implementation mechanism to one with a different binding time. For example, we could introduce a new load-time parameter and *if* statements to replace #ifdef variability (see Sects. 4.1 and 5.3, p. 66 and 110). Such refactoring addresses code smells such as binary chaos and runtime overhead.

In general, we can envision variability-preserving refactorings that change the entire variability-implementation mechanism, for example, rewriting preprocessor-based implementations to plug-ins. Such rewrites should fulfill exactly the behavior preservation requirements of variability-preserving refactorings and could address smells such as #ifdef hell, variability-mechanisms overload, limited extensibility, traceability mess, and binary chaos. However, large-scale rewrites between implementation mechanisms will be difficult to automate and may require significant manual effort.

Example 8.2 As one example of a refactoring that is easy to automate, let us illustrate the merge-features refactoring for feature-oriented programming (see Sect. 6.1, p. 130). In Fig. 8.2, we illustrate two consecutively applied class refinements, called Inner and Outer, introduced by two different feature modules, which we merge into one refinement InnerOuter, located in a single feature module. The two refinements of

```
class Foo {
  void print() {
    System.out.print('<core/>');
  }
}
refines class Foo {
  int a;
  void print() {
    System.out.print('<inner>');
    Super.print();
    System.out.print('</inner>');
  }
}
refines class Foo {
  Object b;
  void print() {
    System.out.print('<outer>');
    Super.print();
    System.out.print('</outer>');
  }
}
```

P = Outer · Inner · Core

```
class Foo {
  void print() {
    System.out.print('<core/>');
  }
}
refines class Foo {
  int a;
  Object b;
  void print() {
    System.out.print('<outer>');
    System.out.print('<inner>');
    Super.print();
    System.out.print('</inner>');
    System.out.print('</outer>');
  }
}
```

P = InnerOuter · Core

Fig. 8.2 Example of the refactoring *Merge Feature*

method print are merged by substituting the Super.print( ) statement in the Outer class by the code of the print method in the Inner class. Merges of other implementation approaches can be automated similarly. □

8.3 Refactoring as Path Toward a Product Line

A special but important use case of automated variability-enhancing refactorings, such as extract feature, is to use them for extractive product-line adoption (Sect. 2.4, p. 39). We can start with a legacy application without variability, and incrementally extract features. In each step, we would preserve the behavior of all products up to this point, but would make more behavior variable. Let us demonstrate such adoption through refactorings by an example.

8.3.1 Example: Extraction of Feature Colored *of the Graph Library*

We demonstrate feature extraction by extracting feature Colored from our graph-library example. Specifically, we want to extract the feature code highlighted in Fig. 8.3 into a separate feature, while preserving the original behavior when this feature is selected. We demonstrate the extraction for feature-oriented programming (see Sect. 6.1, p. 130) and aspect-oriented programming (see Sect. 6.2, p. 141), but other implementation mechanisms could be supported similarly (when using

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Color color = new Color();
  Edge(Node _a, Node _b) {
    a= _a; b = _b;
  }
  void print() {
    Color.setDisplayColor(color);
    a.print(); b.print();
  }
}
```

```
class Color {
  static void setDisplayColor(Color c) {
    ... }
}
```

Fig. 8.3 Intermixed code of the graph example before extraction (feature code highlighted)

a preprocessor, it would even be as simple as just adding conditional-compilation directives around the highlighted code). We use this example also to illustrate the role of granularity during refactorings (see Sect. 3.2.5, p. 59). To make the refactoring understandable and automate, and to ensure that we actually preserve behavior, we break down the refactoring into smaller steps.

First, we introduce a new feature Colored and a corresponding empty feature module into the product line, and mark it as mandatory. Adding a mandatory feature without any implementation clearly does not change the behavior of any existing product.

Second, we move class Color to the feature module. Again, this does not change the behavior of any product, since the class is still included in all products.

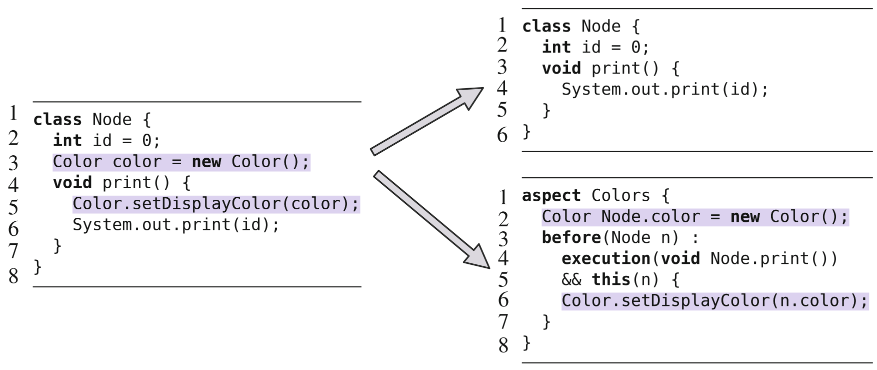
Third, we extract the code from class Node that is related to feature Colored. For feature-oriented programming, we create a new class refinement and move the field declaration of field color there, as shown in Fig. 8.4. Similarly, we move the

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

```
refines class Node {
  Color color = new Color();
  void print() {
    Color.setDisplayColor(color);
    Super.print();
  }
}
```

Fig. 8.4 Extraction of Color code into a Jak refinement



```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

```
aspect Colors {
  Color Node.color = new Color();
  before(Node n) :
    execution(void Node.print())
    && this(n) {
    Color.setDisplayColor(n.color);
  }
}
```

Fig. 8.5 Extraction of Color code into an AspectJ aspect

setDisplayColor call to a method refinement. Note that the base class and the refinement reproduce the original behavior when composed; that is, the extraction is behavior preserving. The solution for aspect-oriented programming is very similar: We create an aspect and move the field to an inter-type declaration and the statement to a piece of advice that reproduces the original behavior when the aspect is woven. In contrast to Jak, we have to be very careful with the use of pointcuts in AspectJ to ensure that we affect only the correct joint points (see the *fragile-pointcut problem* in Sect. 6.2). Corresponding refactorings have been explored and automated in the research communities of feature-oriented and aspect-oriented programming (Schulze et al. 2012; Hanenberg et al. 2003; Monteiro and Fernandes 2005).

After the code of feature Colored was moved to a feature module or aspect, we can mark the feature as optional in the feature model. The behavior remains the same in all feature selections that include feature Colored, but we have enabled generating additional products without that feature. That is, we have performed a variability-enhancing refactoring.

The extraction of feature Colored is straightforward. However, not all extractions are easy. Look at a modified version of method print in Fig. 8.6, which we want to move into feature Colored. In contrast to prior examples, the code to move is spread over the body of method print.

A problem of feature-oriented programming is that the code appears at two places and is not directly addressable by method refinements (a matter of limited granularity of feature-oriented programming, see Sect. 6.1.5, p. 138). As solution, we introduce a *hook method* to prepare an additional extension point which can be refined by a feature module as shown in Fig. 8.8. To create a hook method, we perform a common extract-method refactoring to move the corresponding feature code into its own function. Subsequently, we can then move the feature code to a method refinement of the hook method.

Due to the finer granularity of extensions in AspectJ (see Sect. 6.2.4, p. 149), we can move all feature code from this example to an aspect without the need of an extra hook method, as shown in Fig. 8.8. We can use a combination of call and withincode

```
class Edge {
  Node a, b;
  Color nodeColor = new Color();
  Color weightColor = new Color();
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    Color.setDisplayColor(weightColor);
    weight.print();
    Color.setDisplayColor(nodeColor);
    a.print(); b.print();
  }
}
```

Fig. 8.6 Feature code (highlighted) in the middle of a method

```
class Edge {
  void print() {
    weight.print();
    hook();
    a.print(); b.print();
  }
  void hook() {};
}
```

```
refines class Edge {
  Color nodeColor = new Color();
  Color weightColor = new Color();
  void print() {
    Color.setDisplayColor(weightColor);
    Super.print();
  }
  void hook() {
    Color.setDisplayColor(nodeColor);
  }
}
```

Fig. 8.7 Extracting feature code from the middle of a method using hooks in Jak

```
aspect Colors {
  Node a, b;
  Color Edge.color = new Color();
  before() : execution(void Edge.print()) {
    Color.setDisplayColor(Color.WEIGHTCOLOR);
  }
  after(Edge e) : call(void Weight.print()) &&
    withincode(void Edge.print()) && this(e) {
    Color.setDisplayColor(e.color);
  }
}
```

Fig. 8.8 Extracting feature code from the middle of a method using advice in AspectJ

```
class Graph {
  ...
  Egde add(Node n, Node m, Weight w)
         throws InvalidWeightException {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.setWeight(w);
    return e;
  }
}
```

Fig. 8.9 Feature extraction at the level of method parameters

pointcuts (all calls to method print of class Weight within the code of method print of class Edge) to select the join points to be extended in the middle of the method in question.

Finally, there are examples of code fragments to be extracted for which the granularity of both feature-oriented and aspect-oriented programming is too coarse, such as additional parameters and exceptions, shown in Fig. 8.9. While preprocessors can cope with this fine granularity, in feature-oriented and aspect-oriented solutions, we need to replicate code and provide two separate implementations of the function in question.

8.3.2 Case Study: Refactoring of Berkeley DB with AspectJ

Incremental extract-feature refactorings are a viable approach to extractive product-line adoption, which has been used in several product-line projects. To give additional insights into challenges, advantages, and risks of such adoption, we summarize experience from a refactoring project we performed with AspectJ to extract features from *Berkeley DB*.[1]

Berkeley DB. The subject of the study was the Java version of the database system Berkeley DB (by now acquired by Oracle).[2] Its performance and transaction safety make it popular in open-source and commercial applications. It consists of five large subsystems as shown in Fig. 8.10: access methods as an abstraction and programming interface to the user, a B^+-tree as the internal data storage and index, various caching and buffering mechanisms, a concurrency and transaction system, and a persistence layer.

Already in the Java version that we started from, it was possible to deactivate some parts such transactions at load-time (parameter-based implementation, Sect. 4.1, p. 66), but it was not possible to create a tailored database system at compile time by excluding unnecessary code.

[1] Full details of the study are available in a corresponding publication (Kästner et al., 2007).

[2] http://www.oracle.com/technetwork/products/berkeleydb/

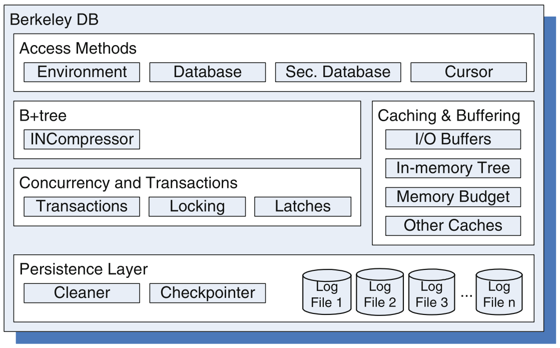


Fig. 8.10 Architecture of Berkeley DB (Java version)

Feature location. By analyzing the domain, manual, configuration parameters, and source code, we identified several candidates for features. The implementations of features were implicit and scattered across the code base. The size of the features varied from small caches to entire transaction or persistence subsystems. All features represented domain abstractions and functionality a user would select or deselect when customizing a database system. From these features, we selected 38 for actual refactoring, as illustrated in the feature diagram shown in Fig. 8.11.

Feature modules. A first interesting issue, which arose early during feature extraction, was *how* to organize the code belonging to a particular feature in terms of aspects. Although it is possible to implement large features in terms of single aspects and even introduce classes as inner classes, it became necessary to decompose large aspects during development to keep them readable (see Sect. 6.3.1, p. 153 and 207 for a discussion of this issue). Hence, we grouped classes and aspects into feature modules, which can be seen as a form of aspectual feature modules (see Sect. 6.3.4, p. 161).

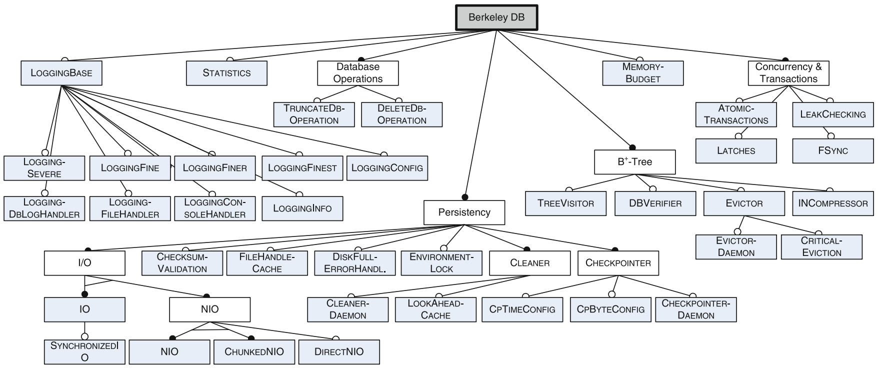


Fig. 8.11 Feature diagram of Berkeley DB after feature extraction

Table 8.1 Refactoring of Berkeley DB: extracted code fragments and their granularity

Refactoring	# times applied
Move method to aspect	365
Extract statement sequence (beginning/end)	214
Move field to aspect	213
Extract statement sequence (with hook method)	164
Extract statement sequence (at call join point)	121
Move class to feature	58
Move interface to feature	4

Feature extraction. To extract features from the original Java code and to encapsulate them in aspects, we used a diverse selection of refactorings, as exemplified in Sects. 8.2.3 and 8.3 and described in more detail in the literature (Hanenberg et al. 2003; Cole and Borba 2005; Monteiro and Fernandes 2005; Binkley et al. 2006).

We needed to extract code at very different levels of granularity, from moving entire classes and interfaces, to extracting entire methods and fields, to extracting individual statements. In Table 8.1, we summarize information on the kind and numbers of refactorings performed. For sequences of statements extracted from a method, we distinguish between (a) statements in the beginning or end of a method, which can be moved to a method refinement or around advice, (b) statements in the middle of a method, which can be reinjected with a call join point (as done in our example in Fig. 8.8), and (c) statements in the middle of a method, for which we needed to introduce a hook method (as done in our example in Fig. 8.7).

Although the refactorings had been documented in the literature for some time, we had to apply them manually, as there was no proper refactoring tool available for this task. We applied feature extraction incrementally, one feature after the other, each into one or multiple (up to 45) aspects and classes. Of the 38 refactored features, 16 were small with less than 140 lines of code, involving 10 or fewer refactorings. The features Latches, Statistics, Logging, and MemoryBudget were large with 958–1864 lines of code, requiring between 118 and 345 refactorings to extract. All other features had a size in-between. Overall, we extracted 10 % of the code base of Berkeley DB into features.

Observations. First of all, we found that product-line adoption by incremental refactoring is possible. We were able to extract code that was previously common to all products or customizable via load-time parameters into aspects, so that different aspect combinations can be woven at compile time for different feature selections.

It is worth noting that this refactoring process just extracted feature implementations in their current form, without any redesign of the system. As a result, the resulting aspects inject fine-grained extensions at many locations, often with the help of hook methods. Although all feature code has been moved to individual aspects (see *separation of concerns* Sect. 3.2.3, p. 55), the implementation does not follow well-defined extension points to hide implementation details (see *information hiding* Sct. 3.2.4, p. 57). As we discussed in the corresponding publications, maintainability of the corresponding code is quite questionable and hardly improves over scattered

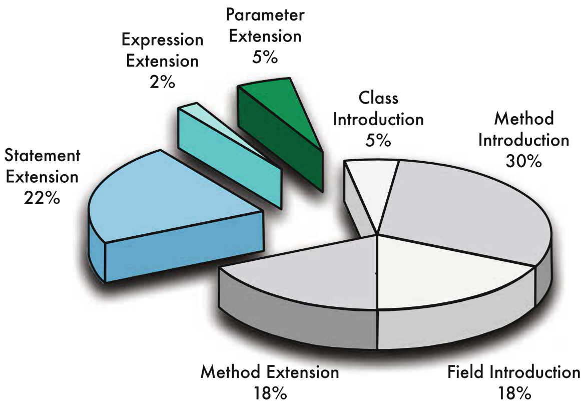


Fig. 8.12 Extensions applied by aspects in the refactored version of Berkeley DB

#ifdef directives (Kästner et al., 2007, 2008a). A proper redesign, possibly defining extension points based on a framework, would have required substantial preplanning (see Sect. 3.2.1, p. 53), but could potentially have resulted in a better maintainable implementation.

As noted already by Clements and Krueger (2002), the extractive adoption approach through refactoring requires tools that can deal with fine-grained extensions. While many extracted code fragments were relatively coarse grained, there were also quite some fine-grained extensions, as illustrated in Fig. 8.12. These fine-grained extensions brought AspectJ to its limits. We often had to resort to workarounds that resulted in complex or 'strange' designs, including many hook methods. An in-depth discussion of the challenges and limitations regarding AspectJ is provided in the original study (Kästner et al., 2007).

With regard to our discussion about the differences between feature-oriented and aspect-oriented programming in Sect. 6.3, the study confirms most observations: Feature extraction in Berkeley DB rarely required crosscutting mechanisms that were unique to AspectJ. Most aspects applied static and basic dynamic crosscutting concerns that could be implemented directly with feature-oriented programming (gray pieces). Homogeneous crosscutting concerns occurred rarely: Less than 11 % of all advice targeted homogeneous crosscutting concerns, and only 2 % advise more than 3 join points.

8.4 Further Reading

A good source to get acquainted with code smells and refactorings in object-oriented programming is the book by Fowler (1999). It is also worth exploring the automated refactorings of development environments, such as Eclipse. For a broader overview, Mens and Tourwé provide a comprehensive survey of existing research in the field of software refactoring (Mens and Tourwé (2004).

Alves et al. (2006) were among the first to propose to extend the traditional notion of refactoring to software product lines. Their view of product-line refactoring aligns with our definition of variability-enhancing refactorings. Initially, they focused mostly on changes to feature models, but later they and their collaborators also incorporated changes to implementation artifacts and the mapping between problem and solution space, also under the names *product-line refinement* (Borba et al. 2010) and *safe evolution* (Neves et al. 2011). Schulze et al. (2012) introduced the notion variant-preserving refactorings, similar to our distinction in Sect. 8.2.2.

A specific variability code smell in product lines is raised by replicated code in feature implementation artifacts. Schulze (2013) discussed this special case in his thesis and presented several refactorings reacting on this code smell. Savolainen et al. (2009) discuss when a product line should preserve mandatory features, even though they do not add anything to the product line's variability.

Thüm et al. (2009) proposed an automated analysis to identify refactorings on feature models (variability-preserving refactorings, excluding also renaming) and generalizations of feature models (equivalent to variability-enhancing refactorings in our definition). We will come back to such analysis in Chap. 10.

Incremental refactorings to extract features were proposed in multiple projects, both using feature-oriented programming (Liu et al. 2006; Rosenmüller et al. 2009a; Schulze et al. 2012) and aspect-oriented programming (Murphy et al. 2001; Hunleth and Cytron 2002; Zhang and Jacobsen 2003; Colyer et al. 2004b; Tešanović et al. 2004; Lohmann et al. 2006a; Kästner et al. 2007). Especially aspect-oriented refactorings have been explored in depth (Hanenberg et al. 2003; Cole and Borba 2005; Monteiro and Fernandes 2005; Binkley et al. 2006). Lohmann et al. (2006a) and Adams et al. (2009) have investigated whether (automatic) refactoring of #ifdef variability into aspects is a viable approach.

Liu et al. (2006) introduced the concept of feature refactoring and extraction. They provided a formal model based on algebra. Their focus lies on dealing with overlapping and interacting features, which we will discuss in the context of feature interactions in Sect. 9.3.

Kästner et al. (2009a) developed a model based on a subset of Java that shows how annotation-based feature implementations can always be transformed to composition-based feature implementations and vice versa (see Sect. 3.1.3, p. 50). The work systematically explores the workarounds needed to rewrite fine-grained extensions.

In many composition-based implementation mechanisms, the order in which features are composed, matters. Apel et al. (2008a) introduced the notion of pseudo-commutativity, whose definition involves reordering of features and a proof that such a refactoring is always possible in AspectJ.

The experience with feature extraction in Berkeley DB using AspectJ is reported in-depth in a separate publication (Kästner et al. 2007). Based on the Berkeley DB case, Kästner et al. (2008a) also discussed the granularity implications of feature implementations. In a parallel effort, Rosenmüller et al. (2009a) refactored the C version of Berkeley DB (using the feature-oriented language FeatureC++). They also discuss implications on performance and binary size.

Exercises

8.1 Select four refactorings from the ‘refactoring’ menu of Eclipse (or another source-code editor). Argue what characteristics these transformations fulfill to be called a refactoring. What code smell does each refactoring address? Demonstrate each refactoring on a suitable code example.

8.2 In Sect. 8.2.1, we have collected an initial list of variability smells.

(a) For each variability smell answer the question of which quality criteria is affected (see Chap. 3 and possibly consider additional criteria). Also, identify in which phase of the product-line development process (see Sect. 2.2) the variability smell is relevant.
(b) Search for instances of each variability smell in the implementation of the graph example in Figs. 4.1 and 5.9 and in the implementation of the chat system (Exercises 4.2, 4.3, 4.5, 4.8, 5.1, 5.4, 6.2, 6.4, and 7.1).
(c) Extend the list with additional variability smells.
(d) Discuss possible product-line refactorings that can eliminate each variability smell.

8.3 Characterize the differences between variability-preserving, variability-enhancing, and product-preserving refactorings. For which purpose can they be used? Provide an example for each from the context of the graph library and your implementation of the chat system (Exercises 4.2ff).

8.4 Describe the product-line refactorings (a) *rename feature* and (b) *merge feature* such that they could be included into a catalog of refactorings and that they could be automated by a tool. Describe the motivation, the mechanics (preconditions, postconditions, transformation steps), their impact on variability, and provide an illustrative example.

8.5 What is the role of refactoring in product-line development, especially, with regard to adoption paths. Exemplify opportunities in the context of scenarios from Exercises 2.9 and 7.7 (p. 44 and 188). Discuss the potential pitfalls of creating a product line by refactorings.

8.6 In Sect. 8.3, we illustrated by an example how to perform a variability-enhancing refactoring to extract a new feature. Perform a similar extraction. If necessary, design suitable refactoring steps. Document and explain all steps.

(a) Extract feature Print (all functionality that is responsible for printing to the command line) from the graph example of Fig. 6.5 (Sect. 6.1) using (i) feature-oriented programming and (ii) aspect-oriented programming.
(b) Extract a feature from the initial implementation of the chat system (Exercise 4.1, p. 96) to make it optional. Use a variability-implementation strategy of your choice.

Chapter 9
Feature Interactions

After reading the chapter, you should be able to

- understand the role of feature interactions in feature-oriented product lines, including intended and inadvertent interactions,
- identify reasons for feature interactions and the feature-interaction problem,
- characterize the nature of 2-way and higher-order interactions,
- outline techniques to detect feature interactions,
- select suitable solutions to implement coordination code for known feature interactions, and
- weigh their mutual strengths and weaknesses.

After a broad discussion of a diverse selection of techniques for implementing features in Part II, we now have a closer look at how features interact when combined with other features. The key idea of feature orientation is to make features explicit in design and code, either by annotating code belonging to a certain feature or by separating and modularizing feature code. But a feature is not an island. Features interact in various ways, both in positive and intended ways, as well as in critical and inadvertent ways. Features are often expected to interact: to exchange information, refine the behavior of other features, reuse the functionality of other features, and accomplish a task in cooperation. However, inadvertent interactions can cause unexpected erroneous behaviors and result in undesired and critical system states. Specifying and managing intended feature interactions as well as detecting and resolving unintended feature interactions is one of the key challenges of feature-oriented product-line development.

In this chapter, we take a closer look at interactions between features and how they manifest in program code and behavior, rather than at the features themselves and their implementations. We illustrate different kinds of feature interactions, discuss strategies to detect them, raise awareness of an instance of the feature-interaction problem, called optional-feature problem, and compare techniques to implement known feature interactions in a controlled manner.

9.1 The Feature-Interaction Problem

A feature that works perfectly well in a given system may exhibit inadvertent behavior when combined with other features. The problem is that, when features are developed independently, it is difficult to predict their mutual interactions when combined. Typically, the behavior of the generated product that contains multiple independently developed features is not easily deducible from understanding the features in isolation—we have to identify and understand their interactions.

Definition 9.1 A *feature interaction* between two or more features is an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved.

An *inadvertent feature interaction* occurs when a feature influences the behavior of another feature in an unexpected way (for example, regarding the expected control flow, program or data state, or visible behavior).

The *feature-interaction problem* is to detect, manage, and resolve (inadvertent) feature interactions among features. □

When features are combined, their interactions need to be coordinated, for example, by ordering their execution, synchronizing data access, defining precedence rules for action handling, and including missing behavior.

For illustration, we provide a list of examples of feature interactions and the corresponding problems:

Example 9.1 Call forwarding and call waiting. A canonical example of a feature interaction occurs in telecommunication networks, in which the two features CallForwarding and CallWaiting interact (Calder et al. 2003). CallForwarding forwards calls made to a busy line to another host. CallWaiting notifies the called party on a busy line of another incoming call and allows the user to switch between both calls. Both features work fine in isolation, but it is unspecified what happens with an incoming call on a busy line if both features are activated. Either feature could take precedence over the other or, even worse, both may attempt to act at the same time.
The interaction between CallForwarding and CallWaiting is undesired and can lead to race conditions and unexpected and inconsistent behavior. The interaction can be hard to predict, because it occurs only in specific conditions (a second call on a busy line). If the interaction is known, we can take measures to control it by giving explicit precedence to one feature (possibly even configurable by the user) or by making both features mutually exclusive (only one can be selected at a time). □

Example 9.2 Fire and flood control. In a building-automation system, as outlined by Kang et al. (2002), feature FireControl activates sprinklers when sensors detect a fire, and feature FloodControl cuts off water supply when water is detected on the floor. Individually, both features operate as desired, but they *interact* in inadvertent

and critical ways: When fire is detected, feature FireControl activates sprinklers; subsequently, feature FloodControl detects standing water, and turns off the water main; as a consequence, the building burns down.
Clearly, the interaction between FireControl and FloodControl is undesired, and inadvertent in the sense that it is hard to predict when planning and implementing the involved features independently. Only when the interaction is known, we can take corresponding steps to manage or resolve it, in this case, for example, by giving explicit priority to feature FireControl over feature FloodControl, controled by some coordination code. □

Example 9.3 Read-only data structures and indexes. As a further example, suppose we incrementally develop a simple data-management solution by starting with a simple read-only data structure and by extending it with two optional features Write and Index. Feature Write adds functionality to add, change, and remove data. Independently, feature Index is developed on top of the read-only data structure to speed-up data retrieval (say, by storing an index as a separate hash map, which is created at load time). Without the other, both features work well on top of the basic data structure; but when combined, changes in the data due to feature Write are not reflected in the index maintained by feature Index. As a consequence, the index can become inconsistent with the data structure such that queries return incorrect results.
Clearly, some coordinating behavior is missing when combining two features. Feature Write does not know about the index that needs to be updated, and feature Index is not aware that the data structure can be modified. When we understand their interaction, we can implement additional coordination code, such that the index is updated properly when the data are modified. □

Example 9.4 Database transactions and statistics. Similar to the previous example, the features Transactions and Statistics interact in a database system. Transactions ensures ACID properties (atomicity, consistency, isolation, durability) in the case of concurrent access to persistent data, and defines the granularity of recovery actions. Statistics collects information for tuning and optimizing data management (for example, number of tables).
Transactions and Statistics interact. On the one hand, Statistics collects information on transaction operations (for example, the number of transactions per second is measured and stored for self-tuning). On the other hand, feature Transactions provides transactional access to statistics data: we want to access data collected by feature Statistics under the umbrella of transactional control.
If we develop both features independently, Statistics would not know about transactions and could not collect statistics on them. Conversely, Transactions would not know about statistics and could not control access to statistics data. Only when we know about this interaction, we can implement corresponding coordination code to make them work correctly together. □

These examples show the breadth of possible feature interactions. Some feature interactions are undesired and inadvertent in the sense that they are hard to predict when planning and implementing features in isolation (see Example 9.2). Other

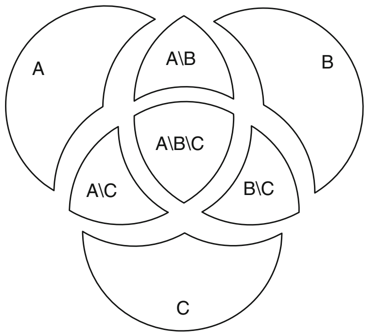


Fig. 9.1 Visualization of three features, three 2-way interactions and one 3-way interactions

feature interactions are desired and planned in advance (see Example 9.4). In any case, features need to be coordinated (more or less explicitly), and that may require additional coordination code, which we discuss in Sect. 9.3 in more depth. That also means, feature interactions may harm feature modularity (see Sect. 3.2.4, p. 57), because, besides the fact that each feature has its own code, there is additional code that does not belong to a single feature, but to a combination. We discuss this issue further in Sect. 9.4.

9.1.1 Higher-Order Interactions

All examples so far illustrate interactions between pairs of features. However, there can also be interactions between more than two features, which are called *higher-order interactions* or *n-way interactions*. The interaction between FireControl and FloodControl is a 2-way interaction or first-order interaction. An interaction that occurs when three features are selected, but not for feature selections of pairs of these features, is called a 3-way interaction or second-order interaction. In Fig. 9.1, we illustrate the possible interactions between the three features A, B, and C by overlapping circles. Three features can give rise to three 2-way interactions and one 3-way interactions (intersections between circles).

Definition 9.2 If n features interact, but none of their strict subsets, this is called an *n-way interaction*. □

Example 9.5 Higher-order interactions are difficult to illustrate in small examples. They emerge from the complex interplay of multiple features. Here, we have created a small but dense code example of a Stack to illustrate a specific case in which three features interact. In this example, variability is encoded using preprocessor directives (see Sect. 5.3, p. 110). The code that coordinates the features of Stack is implemented in the form of nested preprocessor directives (that is, its absence does not cause misbehavior like in the fire-and-flood-control example).

```
 1 class Stack {
 2
 3   boolean push(Object o) {
 4 #ifdef LOCKING
 5     Lock lock = lock();
 6     if(lock == null) {
 7 #ifdef LOGGING
 8       log("lock failed for: "+o);
 9 #endif
10       return false;
11     }
12 #endif
13 #ifdef UNDO
14     rememberValue();
15 #endif
16     elementData[size++] = o;
17     /*...*/
18   }
19
20 #ifdef LOGGING
21   void log(String msg) { /*...*/ }
22 #endif
23 #ifdef UNDO
24   boolean undo() {
25 #ifdef LOCKING
26     Lock lock = lock();
27     if(lock == null) {
28 #ifdef LOGGING
29       log("undo-lock failed");
30 #endif
31       return false;
32     }
33 #endif
34     restoreValue();
35     /*...*/
36 #ifdef LOGGING
37     log("undone.");
38 #endif
39   }
40
41   void rememberValue() { /*...*/ }
42   void restoreValue() { /*...*/ }
43 #endif
44 }
```

Fig. 9.2 Implementing a stack data structure with preprocessor directives; coordination code is implemented by nesting preprocessor directives

In Fig. 9.2, we show the code of the features Locking, Logging and Undo. There is code that belongs to the individual features, and code that belongs to combinations of features (intersections in Fig. 9.1), which is recognizable from nested *#ifdef* directives (the code is only included if multiple features are selected). Note particularly Line 29, which is only included if and only if all three features are selected. If this line caused a failure (for example, threw a null-pointer exception), the failure would occur only in products with all three features, but not in products that select only strict subsets of these features. □

Empirical evidence, for example by Kolberg et al. (2000), Kuhn et al. (2004), and Reisner et al. (2010), indicates that (a) higher-order interactions occur in practice, but also that (b) higher-order interactions are rare compared to interactions between pairs of features; most failures are related to individual features or interactions between two features (such as Examples 9.1–9.4). This empirical evidence suggests to concentrate effort on detecting 2-way feature interactions by analyzing mostly pairs of features, an approach also frequently taken in product-line testing 10.3.2.

9.2 Detecting Feature Interactions

Especially when features are developed independently, detecting and identifying feature interactions is a challenging task, and is still one of the big open problems of product-line development. There are many different strategies, mostly pioneered in the domain of telecommunication systems since the 1990s (Calder et al. 2003; Nhlabatsi et al. 2008), but there is no single strategy that can be claimed as general, scalable, and production-ready, yet.

A key problem of detecting feature interactions is that inadvertent feature interactions may lurk behind any feature combination. In a product line with n features, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of features that may potentially interact, and there are $\binom{n}{k}$ possibilities for k-way interactions. In short, the exponential number of potential interactions limits any systematic and complete search. Even investigating only 2-way interactions (which are empirically much more likely to occur than higher-order interactions) may be overwhelming for industrial-sized product lines.

Within this book, we will not go into details regarding feature-interaction detection, but refer to the corresponding research literature; good starting points are the surveys by Calder et al. (2003) and Nhlabatsi et al. (2008). A typical strategy to detect feature interactions is to make requirements and assumptions regarding features explicit and check them as part of systematic *requirements analysis* in the domain-analysis phase (see Sect. 2.2, p. 19):

- At the requirements level, a typical strategy is to systematically search for shared resources. Two features that share resources may potentially interact over this resource. For example, the features FireControl and FloodControl from Example 9.2 both affect the resource water supply. A typical strategy is to model all resources relevant for each feature and subsequently investigate manually all pairs of features that share a resource.
- The strategy applied to resources can be used also for events (and preconditions of operations). Two features that react to the same event (or that have overlapping preconditions) are potential candidates for feature interactions. For example, the features CallForwarding and CallWaiting from Example 9.1 both react to the same event (that is, an incoming call on a busy line). Again, modeling events allows us to manually investigate all pairs of features reacting to the same event.
- Inconsistent requirements and conflicting goals of features revealed during domain engineering can also be an indicator of potential interactions. For example, features Acceleration to increase the speed of a car and AdaptiveCruiseControl to automatically adjust the distance to other cars by decreasing speed have conflicting goals. Again, requirements and goals need to be made explicit, for example, by modeling them.
- Making assumptions (or invariants) of features explicit can help detecting when an assumption is violated by other features. For example, feature Index in Example 9.3 assumes that the data structure is immutable, an assumption violated by feature Write.

In addition to manual investigation of requirements documents during domain analysis, formal methods can be applied to feature-interaction detection. For example, if preconditions or goals are formally stated, automatic reasoners can detect overlaps, inconsistencies, and critical program states. Similarly, if the behavior of the system can be modeled and assumptions or invariants can be specified formally, model checkers and other reasoners can detect violations.

Example 9.6 Consider the example of an e-mail client that has two optional features for encryption and automatic forwarding (Hall 2005). Both features have been developed and tested based on the basic e-mail client, independently of the respective other feature. As it happens, both features interact in an inadvertent way: The interaction occurs if one host sends an encrypted e-mail to a second host that forwards the e-mail automatically to a third host. If the second host does not have the public key of the third host, it forwards the e-mail in plain text (the forwarding feature has been developed independently and thus does not take encryption into account).

Apel et al. (2013) have shown that this situation contradicts the specification that encrypted e-mails must never be sent in plain text over the network (Hall 2005), and that product-line model-checking technology can be used to detect this situation automatically. □

For a comprehensive overview of formal approaches for feature-interaction detection, see the recent surveys by Calder et al. (2003) and Nhlabatsi et al. (2008). Formal methods have been successfully applied on core models of product lines (Heymans 2012), but to scale them to be able to analyze source code instead of requirements models or manually abstracted models remains an open problem.

Finally, excessive product-line testing can be employed, if suitable test cases are available or if the assumptions of features are specified as run-time assertions. We return to product-line analysis in Chap. 10, including combinatorial testing for feature interactions.

For the remainder of this chapter, we assume that we already know which features interact. We focus on how to implement features with a known interaction by means of coordination code.

9.3 The Optional-Feature Problem

As we have seen so far, interactions between features often require additional coordination code. This code has to be implemented somewhere, and it has to take action only if the corresponding features are present in a given product. The combination of the fact that features can be optional and the need of code to coordinate features give rise to the optional-feature problem.

Definition 9.3 The *optional-feature problem* is the mismatch between intended variability (as specified in the feature model) and the actual variability provided by the implementation, due to coordination code. It occurs when two (or more) optional features interact, and the presence of coordination code reduces the intended variability of the product line. □

Suppose, in our database example, feature Statistics counts the number of transactions per second. If the user configures a database without feature Transactions, the implementation of Statistics breaks (as code concerning transaction management is missing). That is, the *implemented variability* (Statistics requires

```
layer DFS;

refines class Graph {
  void search(Strategy s) { ... }
}
```

```
layer Cycle;

refines class Graph {
  void hasCycles() {
    ...
    search(...);
    ...
  }
}
```

```
layer Weighted;

refines class Edge {
  double weight;
  void setWeight(double w) { ... }
}
```

```
layer ShortestPath;

refines class Graph {
  List shortestPath(Vertex a, Vertex
       b) {
    Edge e1, e2;
    ...
    if (e1.weight > e2.weight) ...
    ...
  }
}
```

Fig. 9.3 Excerpts of the implementations of the features Weighted and ShortestPath of the graph implementation in Jak/AHEAD

Transactions) does not align with the *intended variability* (both features shall be independently selectable).

The optional-feature problem is a specific, but common implementation-level instance of the feature-interaction problem. From a developer's perspective it deserves special attention, because, although often simple to detect, it occurs frequently. When talking about feature interactions so far, we discussed problems regarding incorrect *behavior* due to missing coordination code. In contrast, the optional-feature problem is concerned with incorrect *implementations* of variability that reduce *intended variability* or that have other negative effects such as nonmodular code, as we discuss in the remaining chapter.

Let us illustrate the optional-feature problem further with an example from our graph library.

Example 9.7 The optional-feature problem in the graph example. Suppose a version of our graph example in which the features Weighted and ShortestPath are both optional and independent. The feature model also specifies that feature Cycle conceptually depends on feature DFS.

Now, let us consider implementations of these features. In Fig. 9.3, we show excerpts of implementations using feature-oriented programming using Jak/AHEAD, which are based on the graph implementation of Sect. 6.1. Notice how the implementation of feature Cycle refers to method search from feature DFS, and how the implementation of feature ShortestPath refers to field weight from feature Weighted. Due to these references, there are implementation dependencies from feature Cycle to feature DFS and from feature ShortestPath to feature Weighted.

The implementation dependency between features Cycle and DFS is acceptable and possibly even not avoidable, because the dependency is fundamental in the domain. The feature model already documents this intended dependency.

```
layer BasicGraph;

class Edge {
  double weight = 1;
}
```

```
layer Weighted;

refines class Edge {
  void setWeight(double w) { ... }
}
```

```
layer ShortestPath;

refines class Graph {
  List shortestPath(Vertex a, Vertex b)
          {
    Edge e1, e2;
    ...
    if (e1.weight > e2.weight) ...
    ...
  }
}
```

Fig. 9.4 Alternative implementation of Weighted and ShortestPath in Jak/AHEAD, without any implementation dependency between them

But, the optional-feature problem occurs between the features ShortestPath and Weighted (in this implementation). Although desired, we cannot generate a product for a feature selection with feature ShortestPath, but without feature Weighted: The generated code would contain a dangling reference to field weight. Conceptually, however, such a product should be possible to generate, because both features are optional and independent in the feature model and desired by stakeholders (finding the path with the fewest edges). Hence, we have an implementation dependency that reduces the variability of the product line beyond the domain expert's intention, only because of the coordination code that let the features Weighted and ShortestPath properly interact (Line 27).

Since implementation dependencies are specific to one implementation but not essential in the domain, we can usually provide an alternative implementation without that dependency. We sketch a naive, alternative implementation in Fig. 9.4, in which we can freely combine the features ShortestPath and Weighted: We move the field weight with a default value to the base code, so that feature ShortestPath works independently of feature Weight, without any dedicated coordination code that impairs variability (however, now the base code contains a field that is unused in products without feature Weighted). We discuss different implementation strategies to solve the optional-feature problem in Sect. 9.4. □

As illustrated in the example, the optional-feature problem arises from a mismatch between variability specified in the feature model and variability provided by a specific implementation. The problem occurs when coordination code is hardwired inside a feature. The optional-feature problem often manifests in the form of type errors (for example, a dangling reference to a feature that is absent), which can be detected when actually compiling a derived product (see also Chap. 10 for mechanisms how to detect type errors in all products of a product line).

The optional-feature problem and the feature-interaction problem are highly related, but the goals and challenges are different. In the feature-interaction problem, the challenge is identifying missing behavior (and coming up with corresponding coordination code), whereas, in the optional-feature problem, the challenge is implementing the coordination code such that it does not impair variability.

9.4 Implementing Feature Interactions

After we have identified that two features interact, we need to find a strategy to deal with the interaction, preferably a strategy that does not introduce the optional-feature problem. As one possibility, we can change the feature model to prevent the interaction (or, alternatively, enforce it in all products). For example, we could declare interacting features as mutually exclusive. However, usually the goal is to generate all products properly as intended by domain experts, that is, generate products with both features combined and with each feature in isolation.

Essentially, all resolutions of feature interactions (and the optional-feature problem) can be abstracted to and described by the following pattern: There are two implementations of the individual features, and, to use them together, some coordination code is required to patch up both features (typically, by adding additional code, but potentially also by overriding behavior or removing code). This pattern applies to all examples throughout this chapter.

- *Call forwarding and call waiting (Example 9.1).* When both features are selected, additional code should coordinate them. Coordination code can give priority to one feature, invoke them in sequence, or provide a configuration dialog for users to configure the desired behavior.
- *Fire and flood control (Example 9.2).* When both features are selected, additional coordination code is required to specify that feature FireControl overrules feature FloodControl.
- *Read-only data structures and indexes (Example 9.3).* In the data-structure example, the need for coordination code is especially obvious. When we select both features Write and Index, we need additional code to update the index on write operations.
- *Database transactions and statistics (Example 9.4).* Similar to the previous example, we need coordination code to implement the missing behavior of collecting statistics about transactions and for synchronizing the access to statistics data.
- *Weighted graphs and shortest path (Example 9.7).* Finally, feature ShortestPath can be implemented independently of feature Weighted and vice versa; but, to use them together, we need to include coordination code, such that the shortest-path algorithm uses the correct weights.

For illustration, we use a graphical notation of two interacting features, shown in Fig. 9.5. Each feature is represented by a circle and the overlap between them represents the code that coordinates their interaction. With this graphical notation, we can also illustrate the desired products, as shown in Fig. 9.6: Either we want both features with their interaction properly coordinated, or we want each feature in isolation, without any coordination code. In Fig. 9.1, we already illustrated an equivalent picture for higher-order interactions. Now the question is how to implement feature code and coordination code properly inside a product-line implementation. We will use the graphical notation to illustrate and discuss six implementation strategies.

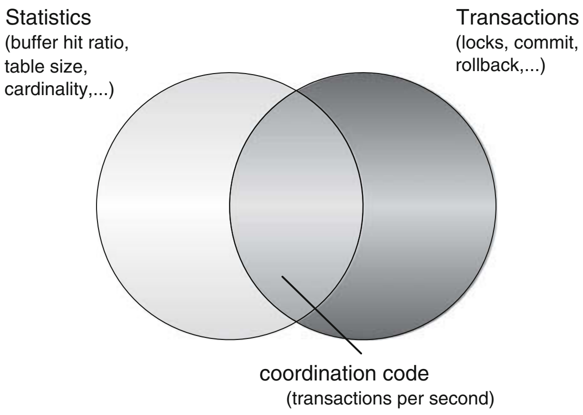

Statistics
(buffer hit ratio, table size, cardinality,...)
Transactions
(locks, commit, rollback,...)
coordination code
(transactions per second)

Fig. 9.5 The features Transactions and Statistics in concert

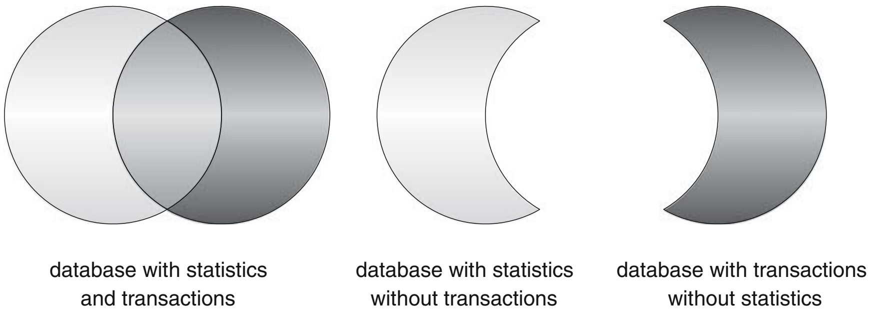

database with statistics and transactions
database with statistics without transactions
database with transactions without statistics

Fig. 9.6 Desired products using the features Transactions and Statistics

9.4.1 Implementation Strategies: Overview and Goals

What makes a good strategy to implement coordination code for feature interactions and to solve the optional-feature problem? There are at least four goals that we want to achieve.

1. *Variability.* The implementation strategy should allow the programmer to generate all products for all feature selections specified as valid in the feature model. That is, we do not want to reduce variability merely due to implementation issues, as described by the optional-feature problem (see Sect. 9.3).
2. *Implementation effort.* The implementation strategy should not require overwhelming implementation effort, because such implementation strategy would not be attractive to use in practice.
3. *Binary size and performance.* The implementation strategy should not increase binary size or decrease performance of products compared to an individual implementation of each product.

4. *Code quality.* Finally, the implementation strategy should not reduce code quality, which would make the product line harder to maintain. As discussed in Part II, there are many trade-offs, but the implementation strategy for interactions should fit to the interaction strategy chosen for features in the first place.

In the following, we discuss six implementation strategies. None of the strategies fulfills all goals; they have different trade-offs, as we will discuss. The discussed strategies are:

- *Change feature model.* Instead of a proper implementation, we exclude problematic feature combinations from the feature model.
- *Multiple implementations.* To account for configurations with and without coordination code, we implement the features separately for each combination.
- *Moving code.* Coordination code is moved to one of the interacting features or to a shared required feature.
- *Conditional compilation.* Using a preprocessor, the coordination code annotated and only complied if both features are present.
- *Optional weaving.* Coordination code is implemented as implicitly optional, using mechanisms inspired by aspect weaving.
- *Distinct module for coordination code.* A distinct module separates coordination code from feature code; the module is automatically included when both features are included.

We discuss these strategies separately, before we compare them in the end.

9.4.2 Change Feature Model

The simplest solution to resolve a known, undesired feature interaction is to *forbid* the problematic feature selection. Instead of solving the problem by adding proper coordination code or reimplementing features, we restrict the feature model to exclude problematic feature selections with an additional constraint. Similarly, we can declare an implementation dependency as domain dependency in the feature model. For example, we could mark the features FireControl and FloodControl from Example 9.2 as mutually exclusive, and we could enforce that feature ShortestPath cannot be selected without feature Weighted in Example 9.7.

This solution, of course, restricts variability (or at least acknowledges the reduced variability) compared to what *should* be valid products in the domain. On the positive side, this solution does not require to change the implementation and, thus, does not affect performance or code quality. Depending on the importance of the excluded products, the reduced variability can be acceptable or can have a serious impact on the strategic value of the product line.

When adopting this solution, we recommend documenting clearly which constraints in the feature model are driven by implementation dependencies. Such documentation helps to separate conceptual considerations in the domain from implementation issues.

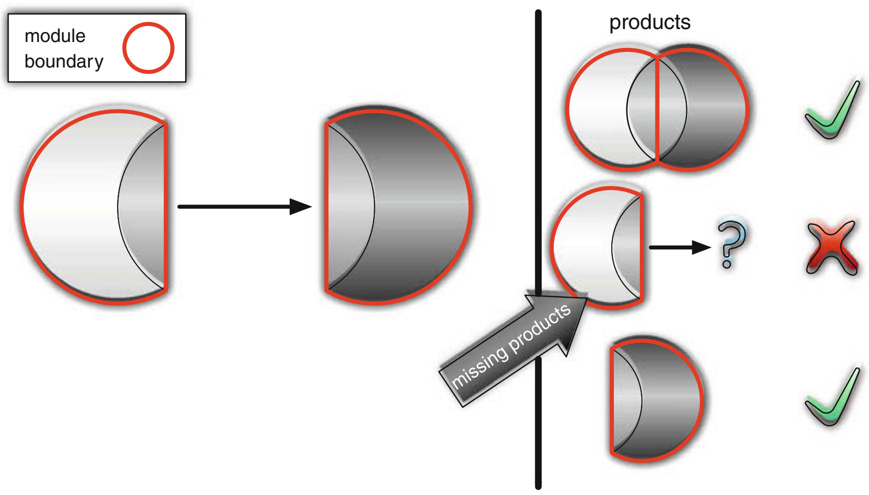


Fig. 9.7 Ignore feature interactions and restrict variability

In Fig. 9.7, we illustrate the solution graphically. We have two features that already contain coordination code, but the coordination code is encoded such that it causes an implementation dependency (as in Example 9.7 about the shortest-path algorithm). Here, we add the implementation dependency to the feature model, again disallowing the corresponding products. In a similar case (not shown graphically), we have two feature implementations but the necessary coordination code is missing (as in Example 9.2 about fire and flood control). We simply declare both features as mutually exclusive in the feature model, thus prohibiting products with both features.

9.4.3 Multiple Implementations

A simple strategy to handle interactions is to provide *multiple implementations* of a feature, one with and one without coordination code. For example, we can have two implementations of feature FloodControl from Example 9.2, one that always turns off water when flooding is detected and one that turns off water only after checking with feature FireControl. Similarly, we could provide an implementation of feature ShortestPath of Example 9.7 for weighted graphs and a second implementation for unweighted graphs. During product derivation, we would then include the suitable implementation, depending on which other features are selected.

Unfortunately, this strategy neglects code reuse, the prime benefit of product-line development, and encourages code replication instead. We need to implement a feature multiple times, one for each feature combination. Furthermore, the approach does not scale if a feature interacts with multiple other features. We need up to 2^{n} implementations of a feature that interacts with n other features.

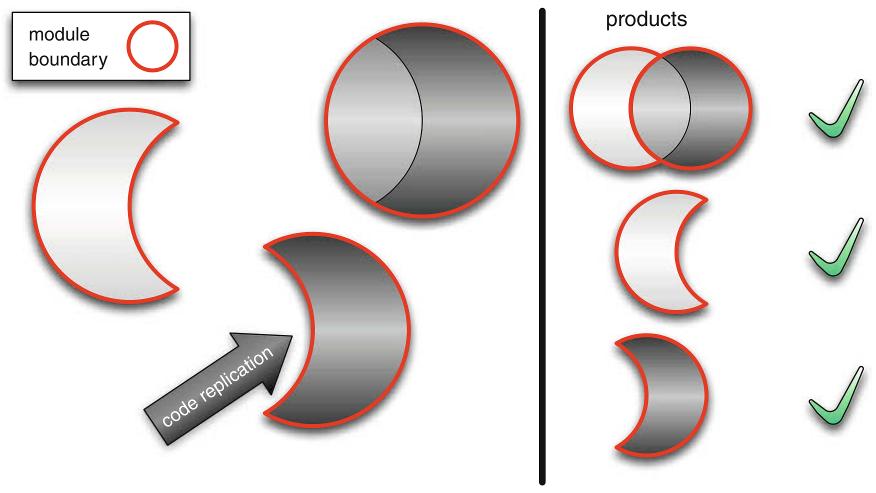


Fig. 9.8 Multiple implementations to make features optional

In Fig. 9.8, we visualize the multiple-implementations strategy. We provide two implementations of the dark-gray feature (say, feature ShortestPath), one with and one without code for coordination with other features. We can generate products for all feature combinations at the price of code replication and additional implementation effort.

9.4.4 Moving Code

In many cases, it is possible to implement the coordination code in one of the two features or in a third feature to which both features refer. For example, feature FloodControl from Example 9.2 could always include code for checking overriding conditions, independent of whether feature FireControl is selected. In the graph example in Fig. 9.4, we have already shown another instance of this solution: We have moved the field weight into the implementation of the base feature. The solution works, because we *move* the coordination code where it does not cause dependencies.

This solution has two drawbacks. The first drawback is a conceptual one: We violate the principle of *separation of concerns* (see Sect. 3.2.3, p. 55), because we move code to implementation units where it does not belong to. For example, feature FloodControl must now be aware of other overriding features as the fire sensor. Also, in our solution for feature ShortestPath in Fig. 9.4, we moved field `weight` that conceptually belongs to feature Weighted into the base code. With this implementation strategy, we give up the clear traceability from features to their implementation, as postulated in Sect. 3.2.2. The point is that coordination code does not belong

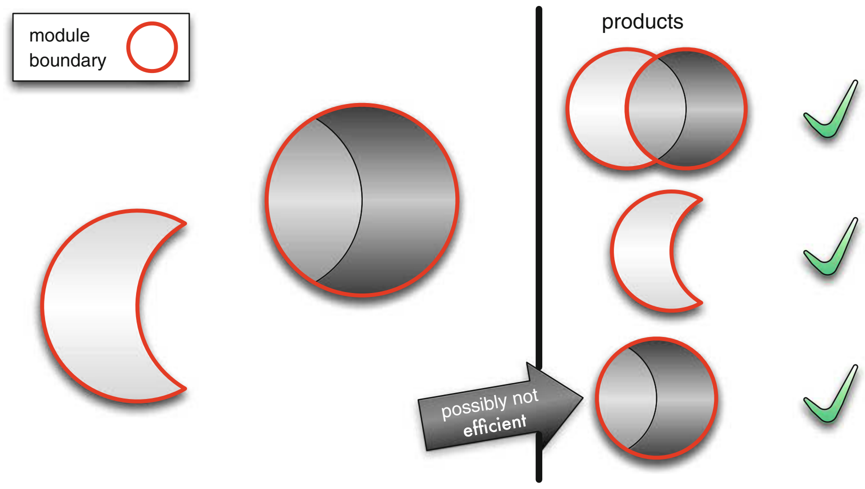


Fig. 9.9 Move code between features

to a single feature, but to combinations of features, so it is between modules that implement individual features.

Second, more technically, we include unnecessary code in some products. In the flood-control example, we would always include and execute code to check for a potential fire sensor, even in products without feature FireControl. In our solution for feature ShortestPath, all edges in all products now contain a field weight (which requires additional memory per object), even when neither feature ShortestPath nor feature Weighted is selected. Including unnecessary code potentially increases binary size, increases memory consumption, and decreases performance.

In Fig. 9.9, we visualize the implementation strategy. The coordination code is part of the implementation of one feature (or of some external feature that both features depend on; not shown here). The coordination code is implemented such that it does not cause a dependency; it remains as dead code in one feature, if the other feature is not selected. As a result, at least one product contains unnecessary code.

9.4.5 Conditional Compilation

If we use an annotative implementation strategy (see *annotation versus composition* 3.1.3, p. 50) for the product line, such as parameters (see Sect. 4.1) or preprocessors (see Sect. 5.3, p. 110), implementing glue code that is only executed if both features are selected is straightforward.

We simply place the coordination code that binds both features in nested *if* statements, nested *#ifdef* directives, or the like. Particularly compile-time approaches (see *binding times* in Sect. 3.1.1, p. 48), such as *conditional compilation* with

```
layer Weighted;

refines class Edge {
  double weight;
  void setWeight(double w) { ... }
}
```

```
layer ShortestPath;

refines class Graph {
  List shortestPath(Vertex a, Vertex b)
          {
    Edge e1, e2;
    ...
#ifdef WEIGHTED
    if (e1.weight > e2.weight) ...
#endif
    ...
  }
}
```

Fig. 9.10 Preprocessor-based implementation of Weighted and ShortestPath that implements coordination code as a conditional block

preprocessors, have the advantage that coordination code is only compiled and included when both features are selected. In Fig. 9.1 (p. 216), we have already illustrated interactions, including higher-order interactions, in terms of nested *#ifdef* directives.

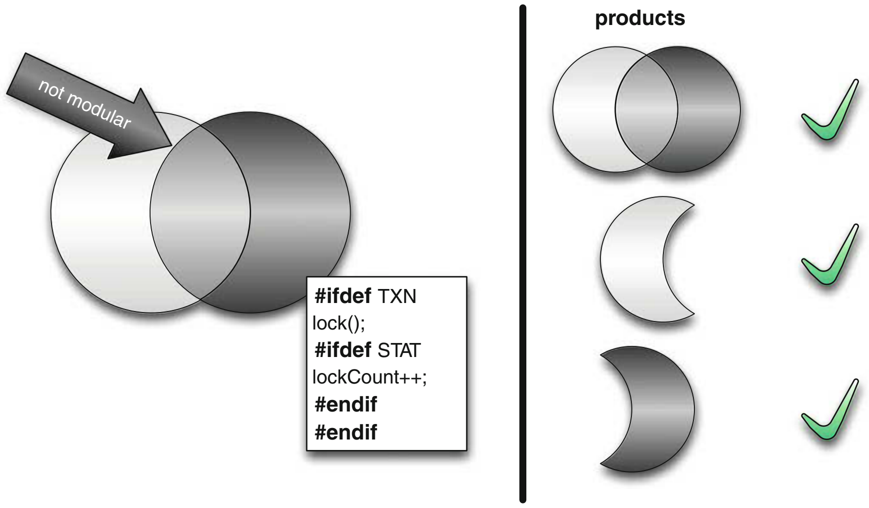
Even when we use a primarily composition-based implementation strategy (such as components, frameworks, feature-oriented programming, and aspect-oriented programming, see Chaps. 4 and 6), we can use *#ifdef* directives inside composition units to conditionally remove unnecessary coordination code before compilation. We illustrate a possible solution for our shortest-path example (Example 9.7) in Fig. 9.10, where we eliminate unnecessary code from the composition unit at compile-time with a preprocessor.

This solution can implement all products without code replication and without compiling unnecessary code causing performance penalties. However, as already discussed in the context of parameters in preprocessors in Sects. 4.1 and 5.3, code quality is usually regarded as poor due to scattering and tangling of feature code and due to neglecting separation of concerns.

The preprocessors solution to known feature interactions is visualized in Fig. 9.11. It does not support an explicit separation of feature implementations. Consequently, in our graphics we have no module borders.

9.4.6 Optional Weaving

Instead of annotating optional coordination code inside the implementation *explicitly* with *#ifdef* directives or similar techniques, several researchers have explored more *implicit* mechanisms (Kästner 2007; Leich et al. 2005; Lohmann et al. 2011). These mechanisms are aimed at composition-based approaches (see *annotation versus composition* in Sect. 3.1.3, p. 50). The mechanisms, which we summarize under the name *optional weaving*, are inspired by the quantification mechanism in aspect-oriented programming (see Sect. 6.2, p. 141).



```
#ifdef TXN
lock();
#ifdef STAT
lockCount++;
#endif
#endif
```

Fig. 9.11 Use of preprocessor annotations for implementing a feature interaction

In aspect-oriented programming, a developer declaratively specifies which code fragments to extend by means of a pointcut. The additional code is woven to all join points matched by the pointcut (possibly one, multiple, or even none).[1] Similarly, optional weaving declares where to add coordination code, but silently fails if the target is not present. As in the conditional-compilation strategy, coordination code is located in one of the features, but only included for compilation when both features are selected.

In Fig. 9.12, we illustrate the idea of optional weaving with a small code example for the fire-and-flood-control example. If the system does not contain methods startFireAlarm and endFireAlarm (when FireControl is not selected), the corresponding glue code in the advice body is simply never woven into the system.

Optional weaving is controversial and has not been fully explored yet. First, the weaving concepts of AspectJ are technically too restrictive for application of optional weaving at larger scale: Pointcuts cannot refer to class names that are possibly not present in the system, and optional weaving is not available for inter-type declarations. However, adopting the optional-weaving idea to other languages seems possible; AspectC is more flexible in this regard than AspectJ (Lohmann et al. 2011). Second, optional weaving depends on the silent failure to weave code when the target is not present. However, silent failure eliminates the chance to check or enforce weaving. For example, if a developer renames method startFireAlarm to beginFireAlarm without updating the pointcut, the coordination code is no longer woven into the system, but this failure is indistinguishable from correctly not weaving the coordination code if feature FireControl is not selected. Critics of optional weaving fear

[1] AspectJ issues a warning for a pointcut that does not match any join point shadow, but does not enforce any specific number of matches.

```
aspect FloodControl {
  boolean isActive = true;

  after(): execution(* *.floodingDetected()) {
    if (isActive)
      ...
  }

  before(): execution(* *.startFireAlarm()) {
    isActive = false;
  }
  after(): execution(* *.endFireAlarm()) {
    isActive = true;
  }
}
```

Fig. 9.12 Example for optional weaving

that the mechanism is too implicit and will result in a large amount of optional code for which it remains unclear when exactly it is applied. However, optional weaving is a comparably recent solution, for which further research is needed.

9.4.7 Distinct Module for Coordination Code

In previous implementation strategies, we have discussed where to move coordination code. An alternative for composition-based implementations is to create yet another *module for code that coordinates features*. As illustrated in Fig. 9.14, we separate coordination code and compose it with the implementation of both features if and only if both features are selected. This strategy also scales for higher-order interactions with more additional modules for coordination code as illustrated in Fig. 9.1.

In the literature, the additional modules for glue code are well known, but have many different names. They are called *lifters* (by Prehofer (1997), because they lift the implementation of one feature to the implementation of another feature), *tiles* (by Kühne (1999), because they connect features from different dimensions shown as a matrix), *derivatives* (by Liu et al. (2006), because they are derived from two features), *connectors* (terminology in Eclipse, because they connect two other plug-ins), and so forth.

In our fire-and-flood-control example (Example 9.2), we could implement both features in separate modules and add the coordination code (which overrides one feature with the other) as a separate module. The separate module is *automatically* included when both features are selected. For our shortest-path example (Example 9.7), we have exemplified one possible solution in Fig. 9.13: The implementation of feature ShortestPath calls a method isLonger with a default implementation that is overridden by coordination code in a separate module (ShortestPath_Weighted).

The key to this implementation strategy is that the additional module for the coordination code is automatically included during generation in the product derivation process, if and only if all participating features are selected. Some automation should

```
layer ShortestPath;

refines class Graph {
  List shortestPath(Vertex a, Vertex b)
          {
    Edge e1, e2;
    ...
    if (isLonger(e1, e2)) ...
    ...
  }
  boolean isLonger(Edge e1, Edge e2) {
    return false;
  }
}
```

```
layer Weighted;

refines class Edge {
  double weight;
  void setWeight(double w) { ... }
}
```

```
layer ShortestPath_Weighted;

refines class Graph {
  boolean isLonger(Edge e1, Edge e2) {
    return e1.weight > e2.weight;
  }
}
```

Fig. 9.13 Alternative implementation of Weighted and ShortestPath with an additional module for the glue code between them

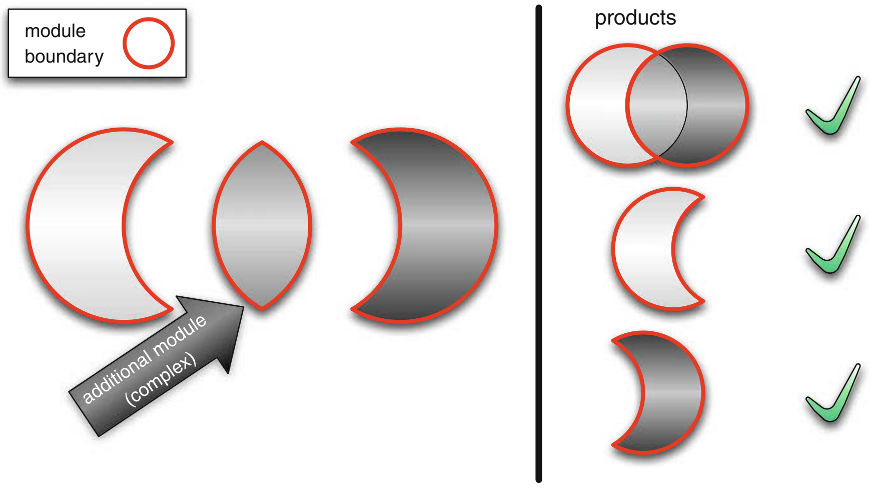


Fig. 9.14 Distinct modules implement coordination code

make sure that we cannot forget the coordination code. In the simplest case, we can create a new feature for the coordination code in the feature model and use constraints to enforce consistent selection (for example, `ShortestPath_Weighted` $\Leftrightarrow$ (`ShortestPath` $\wedge$ `Weighted`)). More sophisticated support in the generation step can help to hide the additional modules. Liu et al. (2006) and Batory et al. (2011) discuss a conceptual and theoretical framework that includes also a concept for naming and automatic selection.

In some cases it can be debatable whether the added module should be hidden and automatically selected, or whether it should be exposed as an extra feature. In our statistics-and-transactions example, we could argue that collecting statistics about transactions is another optional feature, but we could also argue that it belongs conceptually to the features transactions and statistics and should be selected automatically. In most cases, though, the extra module clearly does not constitute a domain abstraction that should be modeled as feature, but mere coordination code that should be included automatically. For example, the coordination code of the fire-and-flood-control example and the write-and-index example should not be offered as optional feature; not including the coordination code when both features are included would be considered as interaction bug in these scenarios.

An interesting insight about this strategy is that it can also work in open-world scenarios (see *software ecosystems* in Sect. 4.3.5, p. 86) where features are provided by independent developers without central authority. For example, in Eclipse, when two plug-ins interact (or should interact), we can add the corresponding coordination code as another plug-in, which is typically called *connector* in this domain. However, a yet unsolved problem in open-world scenarios is to detect the interaction and make sure that a corresponding connector is provided, because there is no central product derivation mechanism that could automatically include the required coordination code.

The use of distinct modules for coordination code is a way of handling interactions, but there are also drawbacks. The number of derivatives may explode in cases where many interactions exist. This may lead to a high number of additional but potentially very small modules that can be overwhelming for developers and hard to understand in isolation. In the future, this increased complexity may be more manageable by tools supporting the automatic refactoring of existing coordination code into distinct modules (see Chap. 8), and their tool-driven maintenance throughout the whole lifetime of a software product line (see Chap. 7)—but more research is needed.

9.4.8 Comparison of Solutions

After we have discussed the implementation strategies in isolation, we can now take a look at the complete picture and discuss how the strategies perform with regard to our four goals of variability, implementation effort, binary size and performance, and code quality.

- Regarding variability, all implementation strategies, except mere changes to the feature model, support the full variability.
- The implementation effort differs significantly. Creating multiple implementations per feature requires significant overhead, and also creating distinct modules for coordination code requires significantly rewriting existing code and creating additional modules.
- Potential overhead regarding binary size, memory consumption, and performance is a problem when moving the code. For optional weaving, we do not yet have sufficient experience.
- Code quality can be discussed controversially. However, code replication of the multiple-implementation strategy is obviously a problem. Also, suboptimal sepa-

Table 9.1 Comparison of implementation approaches for the feature interaction problem

Implementation strategy	Variability	Implementation effort	Binary size and performance	Code quality
Change the feature model		✓	✓	✓
Multiple implementations	✓		✓	
Moving code	✓	✓		✓
Conditional compilation	✓	✓	✓	
Optional weaving	✓	✓	✓?	?
Distinct modules for glue code	✓		✓	✓

ration of concerns and scattering and tangling of code associated with conditional compilation is typically regarded as poor quality. Also, the implicit mechanisms of optional weaving potentially threaten code quality.

We summarize the discussion in Table 9.1.

As the table shows, there is no clear generally preferable strategy. Merely the multiple-implementation strategy seems never to be a good idea. From the remaining strategies, we need to select depending on the context and on which goal is currently most important to developers. Changing the feature model is the easiest solution, but decreases variability. Moving code is also simple, but potentially produces overhead. The main criticism of conditional compilation is its effect on code quality. Creating distinct modules for coordination code seems elegant, but can require significant additional effort from developers. Optional weaving is a new research approach for which not much experience is available. In practice, developers typically will mix and match the approaches according to their needs.

9.5 Experience

From the previous discussion, it seems that developers have to decide for the lesser evil when selecting an implementation strategy for feature interactions. All strategies have different trade-offs and none is without drawbacks. To gain experience, we conducted two case studies on the database systems Berkeley DB (both the Java edition and C edition) and FAME-DBMS (C++ implementation). In all cases, we observed and analyzed product-line development, counted instances of the optional-feature problem, and discussed and explored different implementation strategies. The results were published in Kästner et al. (2009), but here we repeat the key results to provide some context for the different strategies.

The case studies followed different implementation strategies. In the case of Berkeley DB, we decomposed an existing system into a product line. We started with legacy code that already contained many features, without making them explicit or configurable. Interactions between features were already hard-coded (all features were hard-coded as part of the mandatory base code, so was the coordination code between them). We subsequently extracted features and made them optional, a process in which we found coordination code and needed to decide how to implement it. In the case of FAME-DBMS, we developed the product line from scratch. However, since the domain of database systems is well known, we could easily anticipate and plan interactions between features. In both case studies, we focused on the optional-feature problem, that is, how to implement known feature interactions without restricting the intended variability.

9.5.1 Decomposition of Berkeley DB

Oracle's *Berkeley DB*[2] is an open-source database engine implemented in approximately 70,000 lines of code, that can be embedded into applications as a library. In two independent endeavors, we decomposed both the Java and the C version of Berkeley DB into features (described in more detail by Kästner et al. (2007) and Rosenmüller et al. (2008)). We pursued a composition-based implementation strategy with the goal of separating each feature in a distinct module, using aspect-oriented programming with AspectJ in the Java version (see Sect. 6.2, p. 141) and feature-oriented programming with FeatureC++ in the C version (see Sect. 6.1, p. 130).

In the Java version, we identified 38 features. Almost all features are optional and there are only 16 domain dependencies; in theory, we should be able to derive 3.6 billion different products. However, implementation dependencies occurred much more often than domain dependencies. With manual and automated source-code analysis, we found 53 implementation dependencies corresponding to 2-way interactions that were not covered by domain dependencies. We did not find higher-order interactions. We show an excerpt of features and corresponding dependencies between their implementation modules in Fig. 9.15 (implementation dependencies marked with 'x'; there are no domain dependencies between the shown features). Overall, in Berkeley DB, the optional-feature problem occurred between 53 pairs of features, which are independent in the domain, but not in their implementation.

Changing the feature model to simply document all implementation dependencies is not acceptable, because this would restrict the ability to generate tailored products drastically. In pure numbers the reduction from 3.6 billion to 0.3 million possible products may appear acceptable, considering that still many products can be generated. Nevertheless, when having a closer look, we found that especially in the core of Berkeley DB, there are many implementation dependencies. Important features regarding statistics, transactions, memory management, and database operations shown in Fig. 9.15 must be included in virtually every valid feature selection. The remaining variability of 0.3 million products is largely due to several small independent debugging, caching, and IO features. Considering all implementation dependencies, essentially all intended variability is lost.

[2] `http://www.oracle.com/database/berkeley-db`

	11.	12.	13.	20.	27.	29.	30.	31.	33.
11. Atomic Transactions	▨	▨	▨	▨	▨	▨	▨	▨	▨
12. FSync	x	▨	▨	▨	▨	▨	▨	▨	▨
13. Latch	x	x	▨	▨	▨	▨	▨	▨	▨
20. Statistics	x	x	x	▨	▨	▨	▨	▨	▨
27. IN Compressor	x	x	x	x	▨	▨	▨	▨	▨
29. DeleteDbOperation	x		x	x	x	▨	▨	▨	▨
30. TruncateDbOperation	x		x			x	▨	▨	▨
31. Evictor			x	x		x		▨	▨
33. Memory Budget	x		x	x		x		x	▨

Fig. 9.15 Implementation dependencies ('x') in Berkeley DB (excerpt)

In the C version, which has a very different architecture and was independently decomposed into a different set of features, we identified 24 features (see Rosenmüller et al. (2008) for details). But, the overall picture is similar: With only 8 domain dependencies almost 1 million products are conceptually possible, but only 784 products can be generated considering all 78 implementation dependencies between feature pairs that we found. Again, important features were de facto mandatory in every feature selection.

These numbers give a first insight into the impact of the optional-feature problem. We found more instances of feature interactions than there are features (as explained in Sect. 9.2, there can be a quadratic number of interactions between pairs of features, and even an exponential number considering also higher-order interactions). In Berkeley DB, the strategy to merely change the feature model reduces variability to a level that makes the product line almost useless.

Exploring Implementation Srategies

After the analysis revealed that changing the feature model is not a general option, we explored different solutions to eliminate implementation dependencies. Focusing on a clean composition-based implementation, and following the principle of separation of concerns, we started with creating distinct modules for coordination code.

In the Java version, we first created nine distinct modules to encapsulate coordination code of all nine interactions of the feature Statistics. These nine modules alone required over 200 additional pieces of advice or inter-type declarations with AspectJ. Of 1867 lines of code of the statistics feature, we rewrote 76 % as modules (which would also be the amount of code we needed to move into different features for the moving-code strategy). This shows that most of the functionality of feature Statistics is in its interactions with other features. In the C version, we created 19 distinct modules for coordination. In both versions, we experienced the necessary rewrites as rather tedious. We needed between 15 min and 2 h for each new module, depending on the amount of code. Due to the high effort, we refrained from creating distinct modules for all implementation dependencies.

Next, we experimented with conditional compilation. In the C version, we used *#ifdef* statements inside FeatureC++ modules, as illustrated in Sect. 9.4.5. In the Java version, we used a preprocessor-like environment *CIDE* to eliminate all implementation dependencies (see *virtual separation of concerns* in Sect. 7.4 p. 184). Using conditional compilation was significantly faster than implementing distinct modules, because no changes to the code were necessary except for introducing annotations. However, we deviated from our original goal of a clean composition-based implementation. As a result, feature code is scattered and tangled, with up to 300 annotated code fragments in 30 classes per feature.

In Berkeley DB, both creating distinct modules for coordination code and conditional compilation were acceptable despite their drawbacks. While we prefer a clean separation of concerns, we felt that the required effort was overwhelming. In this project, a mixture of additional modules and conditional compilation felt as a good compromise to us.

9.5.2 Design and Implementation of FAME-DBMS

The question remains of whether the high number of implementation dependencies is caused by the design of Berkeley DB and our subsequent refactoring, or whether they are inherent in the domain. In the latter case, they should also appear in a database product line that was designed from scratch.

FAME-DBMS is a prototype of a database product line implemented with FeatureC++ (see *feature-oriented programming* in Sect. 6.1 p. 130). FAME-DBMS was designed specifically for small embedded systems. Its goal was to show that product-line technologies are appropriate to tailor data management for special tasks in even small embedded systems (for example, BTNode with Nut/OS, 8 MHz, and 128 kB of memory). FAME-DBMS is minimalistic and provides only essential data management functionality to store and retrieve data using an API. Advanced functionality such as transactions, set operations on data, or query processing was not part of the prototype. The initial development that we describe here was performed in a project by a group of four graduate students at the University of Magdeburg, after our experience with Berkeley DB.

Design

FAME-DBMS was designed after careful domain analysis and analysis of scenarios and existing embedded database engines. The initial feature model of FAME-DBMS as presented in the kick-off meeting of the project, is depicted in Fig. 9.16 (only layout and feature names were adapted for consistency). It contains 14 concrete features (grayed features were not linked to code). To customize FAME-DBMS, we can choose between different operating systems, between a persistent and an in-memory database, and between different memory-allocation mechanisms and paging strategies. Furthermore, index support using a B^+-tree is optional, so is debug logging, and finally it is possible to select from three optional operations *get*, *put*, and *delete*.

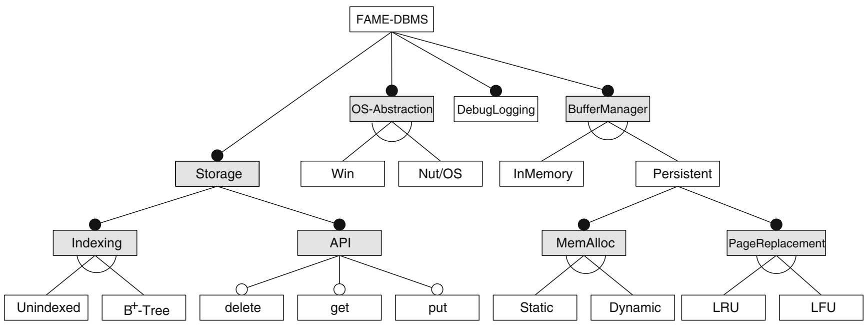


Fig. 9.16 Initial feature model of FAME-DBMS

Fig. 9.17 Domain dependencies ('o') and implementation dependencies ('x') in FAME-DBMS

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Nut/OS														
2. Win	o													
3. InMemory														
4. Persistent			o											
5. Static	x	x	o	o										
6. Dynamic	x	x	o	o	o									
7. LRU			o	o										
8. LFU			o	o			o							
9. Unindexed														
10. B^+-tree									o					
11. Put	x	x		x					x	x				
12. Get	x	x							x	x	x			
13. Delete	x	x		x					x	x	x	x		
14. Debug	x	x	x	x	x	x	x	x	x	x	x	x	x	

The intended variability, captured by the feature model, describes 320 valid feature selections.

Soon after the initial design, the students realized that many of the features would require code to coordinate interactions. In Fig. 9.17, we show the domain dependencies ('o') and the implementation dependencies ('x') that were expected in addition. The latter give rise to the optional-feature problem. For example, the debug logging feature has an implementation dependency with every other feature (it extends them with additional debugging code), but should be independent according to the feature model. Also the features Get, Put, Delete, Nut/OS, and Win interact with many other features. This analysis already shows that it is necessary to find suitable strategies for the implementation of coordination code. Again, merely changing the feature model was not an option, because this would almost entirely eliminate variability.

Implementation

We left the implementation up to the students. We recommended the solution with extra modules for coordination code, but did not enforce it. In the remainder of this section, we describe the final implementation at the end of the project and discuss choices and possible alternatives.

First, as expected, the students chose implementation strategies to implement coordination code without restricting variability. There were only two exceptions, in which they changed the feature model: They merged features Put and Delete, so they cannot be selected independently, and they marked feature Get as mandatory. With this choice, they reduced the number of interactions they needed to implement from 36 to 22. At the same time, they reduced the number of possible products from 320 to 80. The intension behind changing the feature model was the following: Although there are use cases for a database that can write but not delete data, or even for a write-only database (see Szewczyk et al. (2004)), these feature selections are so rarely demanded that the students considered the reduced variability acceptable.

Second, the 11 interactions of feature DebugLogging with other features have been implemented using conditional compilation. This scattered the debugging code across all implementation modules. Alternatively, some debugging code could have been moved into the base implementation causing a small run-time penalty, or 11 additional modules could have been created. The students decided to use conditional compilation despite the scattered code, because it required least effort.

Third, the implementation of feature B^+-tree always contains code to add and delete entries, even in read-only configurations. This is an instance of the moving-code implementation strategy. In read-only configurations with feature B^+-tree, the additional code is included but never called. The students choose this strategy, because it was simpler than creating distinct modules. An ex-post evaluation revealed that the unnecessary code increased binary size by 4–9 kB (5–13 %; depending on the remaining feature selection).

Fourth, the remaining 10 interactions were implemented using distinct modules, following our original recommendation. The students considered the additional effort as the lesser evil compared to a further reduction of variability, a further scattering of code with preprocessor annotations, or a further unnecessary increase in binary size. The multiple-implementations strategy was not considered at any time.

The implementation of FAME-DBMS used a combination of various strategies, but still increased the code size of some products, reduced variability, and required effort for creating 10 additional modules. Even in such small product line, the optional-feature problem pervades the entire implementation.

All our case studies are from the database domain and we believe that other developers may have chosen different trade-offs. The frequent occurrence of the optional-feature problem may be due to the domain or the used fine granularity, but we believe that observations will be possible in other product lines. In each case, developers have to make their own choices with regard to implementing coordination code, but we hope that sharing our experience helps with these trade-offs.

9.6 Further Reading

When the feature-interaction problem became a crisis in the telecommunications industry in the late 1980s (Bowen et al. 1989), researchers began to develop formalisms to enable automatic detection of feature interactions (Blom et al. 1994; Bruns et al. 1998; Felty and Namjoshi 2003; Lin and Lin 1994; Pomakis and Atlee 1996), architectures that avoid classes of interactions (Hay and Atlee 2000; Jackson and Zave 1998; Utas 1998; van der Linden 1994; Zave 2010), and techniques for resolving interactions at run-time (Griffeth and Velthuijsen 1994; Tsang and Magill 1998).

While the pioneering work on the feature-interaction problem in telecommunication systems was foundational and successful (see the surveys by Calder et al. (2003) and Nhlabatsi et al. (2008)), it is limited, as it is based on assumptions that hold for telecommunication systems, but not for other domains, for example, the enforcement of architectural styles or the need of explicit specifications of feature interactions.

Recently, researchers began to propose solutions (mostly based on verification techniques) to the feature-interaction problem that take the specific properties of software product lines into account, especially, the possibly exponential number of products (Apel et al. 2013; Classen et al. 2012; Lauenroth et al. 2009; Thüm et al. 2012).

The testing community has introduced the concept of interaction faults, which denotes implementation defects that are only triggered when multiple parameters are set to specific values. Garvin and Cohen (2011) provide a good definition of feature-interaction faults, which includes the possibility that a defect occurs only when a feature is *deselected* in combination with another feature. Several researchers have empirically investigated how defects and code paths in practical software systems are related to feature interactions (Kolberg et al. 2000; Kuhn et al. 2004; Reisner et al. 2010). The general insight is that the majority of bugs and code paths are triggered by individual features or n-way interactions with low values for n. This confirms a tendency in the testing community to search primary for 2-way interactions, in software product lines and any other software systems.

Our discussion of the optional feature problem and the trade-offs of different implementation strategies, as well as our experience report, is based on prior work (Kästner et al. 2009). Most implementation strategies are quite obvious and not discussed in-depth in the literature. However, the strategy of using additional modules was discussed implicitly and explicitly in many contexts (Kühne 1999; Liu et al. 2006; Lopez-Herrejon et al. 2005; Prehofer 1997). Also, optional weaving has been discussed repeatedly in recent years (Adler 2010; Kästner 2007; Leich et al. 2005; Lohmann et al. 2011).

Exercises

9.1. Collect a list of interactions that may appear between (a) common features of a mobile phone and (b) features identified in Exercise 2.4 (p. 43).

9.2. Hall (2005) has collected a list of possible interactions between basic features of an e-mail delivery system, of which we show two below. Discuss possible strategies how these two interactions could have been detected.

(a) Bob sends a signed message to Alice, who has no signing key provisioned. Yet Alice forwards the message to a third party. The message will arrive there signed, not by the sender (Alice), but by the originator (Bob). Thus, the signature will not verify, even if the third party has a verifying key for Bob, since the verification is defined to determine whether the message was signed by the sender of the message.

(b) Bob sets up forwarding to Alice. Alice has an auto-response feature enabled. A third party sends a message to Bob, which is forwarded to Alice. The auto-response is sent back to Bob and then forwarded to Alice. Thus, messages arriving at Alice via Bob are not effectively auto-responded.

9.3. Are all feature interactions undesired? Discuss this issue by means of the following feature interaction in a phone system:[3] Alice is forwarding calls to Bob, and Bob is forwarding calls to Carol. If Alice is called, should the call be forwarded to Carol?

9.4. Extend the graph library with an additional feature that introduces a feature interaction. Discuss the implementation strategies from Sect. 9.4; argue which implementation strategy is most suitable to resolve the optional-feature problem in this case.

9.5. Did any implementation of the chat system (Exercise 4.1 and following) give rise to feature interactions or the optional-feature problem? If yes, how did you handle the interaction?

Below is a list of extensions for the chat system. For each extension:

(a) Explain which interaction is triggered by the extension. What is the required coordination code in this case (if any)?
(b) Modify the chat system of Exercise 4.5, 6.2, or 6.4 accordingly.
(c) Observe whether the extension triggers an instance of the optional-feature problem. Illustrate the optional-feature problem in terms of intended and actual variability.
(d) Explore and compare all implementation strategies discussed in Sect. 9.4. Argue which implementation strategy is most suitable for this extension.

The extensions for this exercise are:

1. Document in the history of the server whenever a user tries to authenticate with an incorrect password (features History and Authentication).

[3] Adopted from: `http://www2.research.att.com/~pamela/faq.html`.

2. Ensure that authentication messages are encrypted and that the spam filter always works on decrypted messages (features Encryption, Authentication, and SpamFilter from Exercise 4.5, p. 96)
3. Ensure that a message is never encrypted twice (encryption features).
4. The dialog showing the history should display the color of the message (features History and Color).
5. Even when the user is busy, messages with red color are still delivered (features Color and BusyStatus from Exercise 6.4, p. 173).

9.6. Consider a product line with 20 optional features of which 10 participate in 2-way interactions (each feature participates only in one interaction).

(a) How many modules are necessary to implement all coordination code using the distinct-modules strategy?
(b) If we change the feature model to forbid interacting feature to be selected independently, how many valid products can be generated?

Chapter 10
Analysis of Software Product Lines

After reading the chapter, you should be able to

- characterize opportunities and challenges for analyses of product lines (feature model, implementation artifacts, mappings),
- perform analyses of feature models using a corresponding encoding as satisfiability problem,
- detect dead code fragments manually and mechanically, and
- outline and compare strategies to perform variability-aware type checking of entire product-line implementations.

Variability raises new challenges for establishing correctness or any kind of functional or nonfunctional guarantees about programs. Testing, type checking, static analysis, verification, or software and performance measurement are well-established for individual systems, but they do not scale to product lines due to the huge configuration space with a combinatorial explosion of feature selections.

Instead of a single product, a product line gives rise to dozens, thousands, or billions of potential products that we might want to analyze. Analyzing every product in isolation, using traditional analysis methods in a brute-force fashion, will not scale: For a product line with n optional features, there are up to 2^n products for distinct feature combinations. Already with 33 optional and independent features, we could create a product line with more products than persons on the planet; and from a product line with 265 optional and independent features, we could derive more products than there are estimated atoms in the universe.[1] Industrial product lines often have even more features; for example, according to Refstrup (2009), HP's product line *Owen* of printer firmware has roughly 2,000 features and the *Linux kernel* currently has over 10,000 features (Tartler et al. 2011). These numbers clearly rule out any brute-force strategy for product-line analysis.

Traditionally, developers get away with analyzing only a small set of products. For example, Refstrup (2009) reports that even though HP's printer firmware has

[1] There are estimated $7B \approx 2^{33}$ people on earth and $10^{80} \approx 2^{265}$ atoms in the universe.

2,000 features, HP's developers derive and test firmware only for about 100 current printer models. Only when they produce a new printer, they test its (new) feature combination; when a printer is no longer supported, the corresponding firmware is no longer derived and tested. However, the strategy of checking few selected products works only in cases when few products of a product line are actually needed and application engineering is performed by the original developers.

In contrast, our view of feature-oriented product lines includes scenarios in which users can freely configure features and automatically generate the corresponding product. For example, instead of choosing from a small set of preconfigured products, users of the Linux kernel can freely select from the 10,000 features they want to include in their kernel. In such scenario, the Linux developers cannot predict which products need testing; users may select any product and expect it to work properly.

In this chapter, we discuss a broad range of strategies and methods to analyze a *whole* product line (or to attain a reasonable coverage) instead of analyzing all derivable products individually. To this end, we explicitly consider variability in the analysis, hence the name *variability-aware analysis* (also sometimes named *product line-aware analysis*, *family-based analysis*, *feature-aware analysis*, *whole-product-line analysis*, or *150-% analysis*). We introduce mechanisms that are specific to product-line variability and illustrate how to extend existing mechanisms such as type checking, model checking, and static analysis to cover entire product lines.

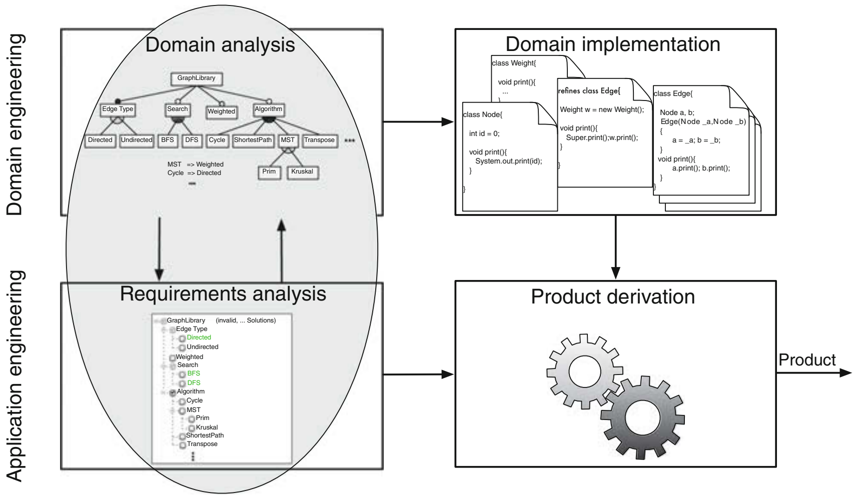
We start with the basic analyses of feature models (Sect. 10.1) and a simple analysis of the mapping between features and implementation artifacts (Sect. 10.2). Lastly, we discuss examples of how to lift existing analyses to entire product lines (Sect. 10.3).

10.1 Analysis of Feature Models

Analyzing feature models is a good starting point, because it is well understood and comparably simple. These analyses not only provide support for reasoning about feature models themselves, but also provide a foundation for analyzing the code of a software product line later. All the analyses discussed in this section are concerned with the feature model (in domain analysis) and feature selections (in requirements analysis) as illustrated in Fig. 10.1.

Among many others, feature model analyses can provide answers to the following questions:

- Is a given feature selection valid for a given feature model?
- Is the given feature model consistent (that is, is there at least one valid feature selection)?
- Do the following assumptions hold for my feature model (testing)?
- Which features are mandatory?
- Which features can never be selected (dead features)?
- How many valid feature selections does a given feature model have?
- Are two feature models equivalent (that is, do they define the same feature selections)?



```
class Weight{

  void print(){
    ...
  }
```

```
class Node{

  int id = 0;

  void print(){
    System.out.print(id);
  }

}
```

```
refines class Edge{

 Weight w = new Weight();

 void print(){
   Super.print();w.print();
 }

}
```

```
class Edge{

  Node a, b;
  Edge(Node _a,Node _b)
  {
      a = _a; b = _b;
  }
 void print(){
      a.print(); b.print();
  }
}
```

Fig. 10.1 Analysis of feature models in domain and application engineering

- Given a partial feature selection, what other features must be included (or excluded)?
- Given a partial feature selection, what features should be selected to produce the product with lowest cost, lowest size, best security, or highest performance?

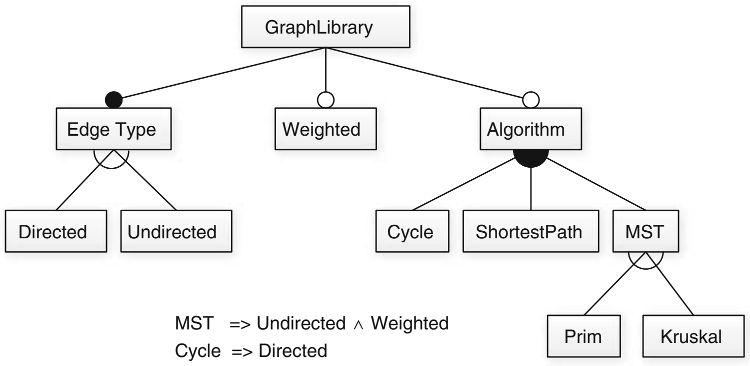
All of these questions can be answered with analyses of feature models and feature selections, and can be automated with tool support. Each can be encoded as formula in a suitable formalism, and automated solvers can answer the questions more or less efficiently. In this chapter, we discuss encodings as Boolean satisfiability problem that can be answered with SAT solvers, but other encodings and tools are possible.

A word on notation: We denote the set of all features of a product line with $\mathtt{F}$ and the set of all possible feature selections by $2^{\mathtt{F}}$. We denote the propositional representation of a feature model as ϕ. We write $\models \mathtt{p}$ to denote that formula $\mathtt{p}$ is a tautology. We write SAT($\mathtt{p}$) to determine whether formula $\mathtt{p}$ is satisfiable (has at least one model).[2] Both notions are translatable: $\models \mathtt{p} \;\equiv\; \neg\mathrm{SAT}(\neg \mathtt{p})$.

10.1.1 Valid Feature Selection

A question that we can answer easily is whether a given feature selection is valid for a given feature model. To this end, we translate the feature model into a propositional formula ϕ as described in Sect. 2.3.3. A feature selection is *valid* if and only if the interpretation of the formula, in which we assign `true` ($\top$ for short) for every

[2] A *model* is a solution (that is, a true/false assignment to each feature variable) that satisfies ϕ.



$$\begin{aligned}\phi =& \texttt{GraphLibrary} \land \texttt{EdgeType} \land (\texttt{Directed} \lor \texttt{Undirected}) \land \neg(\texttt{Directed} \land \texttt{Undirected})\\ &\land ((\texttt{Cycle} \lor \texttt{ShortestPath} \lor \texttt{MST}) \Leftrightarrow \texttt{Algorithm}) \land (\texttt{Cycle} \Rightarrow \texttt{Directed})\\ &\land ((\texttt{Prim} \lor \texttt{Kruskal}) \Leftrightarrow \texttt{MST}) \land \neg(\texttt{Prim} \land \texttt{Kruskal}) \land (\texttt{MST} \Rightarrow (\texttt{Undirected} \land \texttt{Weighted}))\end{aligned}$$

Fig. 10.2 Simplified feature model of our graph example

selected feature and `false` ($\bot$ for short) otherwise, it is a model of the formula. In other words, we substitute every variable corresponding to a selected feature by `true` and every other variable by `false`; the selection is valid if ϕ is `true`. The operation is computationally very cheap (linear in the size of ϕ).

Example 10.1 In Fig. 10.2 below, we show a subset of the feature model of our graph example (originally Fig. 2.6, p. 33) and its corresponding propositional formula ϕ.

To check whether {GraphLibrary, EdgeType, Directed} is a valid selection, we substitute all variables of ϕ with the corresponding assignment:

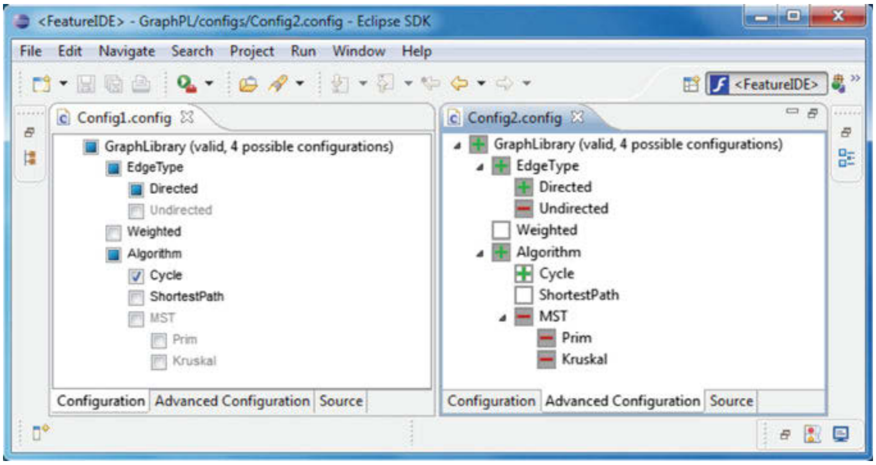
$$\begin{aligned}\phi =& \top \land \top \land (\top \lor \bot) \land \neg(\top \land \bot)\\ &\land ((\bot \lor \bot \lor \bot) \Leftrightarrow \bot) \land (\bot \Rightarrow \bot)\\ &\land ((\bot \lor \bot) \Leftrightarrow \bot) \land \neg(\bot \land \bot) \land (\bot \Rightarrow (\bot \land \bot))\\ =& \top\end{aligned}$$

This result confirms that the selection is valid.

At the same time, {GraphLibrary, EdgeType, Directed, Undirected} is not a valid selection:

$$\begin{aligned}\phi =& \top \land \top \land (\top \lor \top) \land \neg(\top \land \top)\\ &\land ((\bot \lor \bot \lor \bot) \Leftrightarrow \bot) \land (\bot \Rightarrow \bot)\\ &\land ((\bot \lor \bot) \Leftrightarrow \bot) \land \neg(\bot \land \bot) \land (\bot \Rightarrow (\bot \land \bot))\\ =& \bot\end{aligned}$$

□


<FeatureIDE> - GraphPL/configs/Config2.config - Eclipse SDK
File Edit Navigate Search Project Run Window Help
<FeatureIDE>
Config1.config
Config2.config
Configuration
Advanced Configuration
Source

Config1.config:
- GraphLibrary (valid, 4 possible configurations)
  - EdgeType
    - Directed
    - Undirected
  - Weighted
  - Algorithm
    - ✓ Cycle
    - ShortestPath
    - MST
      - Prim
      - Kruskal

Config2.config:
- GraphLibrary (valid, 4 possible configurations)
  - EdgeType
    - Directed
    - Undirected
  - Weighted
  - Algorithm
    - Cycle
    - ShortestPath
    - MST
      - Prim
      - Kruskal

Fig. 10.3 Feature-selection dialog in FeatureIDE with an incomplete feature selection (simple variant left, advanced variant right)

A typical application of this analysis is during requirements-analysis phase of application engineering. When a user selects features, the tool can give immediate feedback whether the current selection is valid. For example, in Fig. 10.3, we show a screenshot of the configuration dialog of FeatureIDE.[3] Next to the root feature, FeatureIDE indicates whether the current selection is valid. In this example, the current selection is invalid because the user has not yet selected feature Directed or Undirected. (Both Directed and Undirected are `false` in this evaluation).

10.1.2 Consistent Feature Models

The next question we attempt to answer is: Is there *any* valid feature selection for a given feature model? We say a feature model is *consistent* if it has at least one valid feature selection; otherwise, we say the feature model is *inconsistent*. In a model with many cross-tree constraints, such question is not trivial to answer; adding contradictory constraints by accident can easily happen.

Naively, we could automatically check all possible feature selections ($\mathtt{s} \in 2^{\mathtt{F}}$, exponentially many) until we find a valid one. In practice, we encode the question as a Boolean satisfiability problem and use a SAT solver to compute the answer. To ask whether a feature model is consistent, we simply determine whether its Boolean representation ϕ is satisfiable ($\mathrm{SAT}(\phi)$). Modern SAT solvers are mature tools, which

[3] For more information on the open-source tool FeatureIDE, see Appendix A.

can solve such problems with great efficiency.[4] If desired, most SAT solvers can also output a valid feature selection (that is, a model of the formula).

Determining whether a propositional formula is satisfiable is an NP-complete problem, meaning that there is no guaranteed efficient algorithm. Consequently, determining whether a feature model is consistent is NP-complete, as well. We can verify a solution quickly (see Sect. 10.1.1), but there is (most likely) no polynomial algorithm to check whether a solution exists; in the worst case, execution time might be exponential in the number of features. Despite exponential worst-case complexity, researchers have empirically shown that modern SAT solvers can solve practical Boolean satisfiability problems quickly in the context of feature-model analysis, even for very large feature models (Mendonça et al. 2009). For real-world feature models, modern SAT solvers, such as SAT4J (Berre and Parrain 2010), can determine satisfiability within milliseconds on good-sized formulas.

Example 10.2 The feature model depicted in Fig. 10.2 is consistent. In Example 10.1, we showed that at least one valid feature selection exists. In contrast, if we extended the feature model as follows: $\phi' = \phi \wedge (\mathsf{Directed} \wedge \mathsf{Undirected})$, ϕ' is inconsistent and has not a single valid feature selection. Reason: ϕ' requires that both features Directed and Undirected be selected and not selected together. □

10.1.3 Testing Facts about Feature Models

A domain engineer typically knows certain facts that must hold in the domain and that should also hold in the feature model. For example, in graph library of Fig. 2.6 (p. 33) we know that feature Cycle requires feature Directed. This fact must be embodied in the feature model; the feature model must not allow any feature selection to violate that dependency. In the graph example, the constraint is obviously fulfilled, it is even stated directly as a cross-tree constraint in the feature model. However, not all constraints may hold so obviously, especially in large models.

To test a feature model, we check an assumption, encoded as propositional formula ψ (such as $\mathsf{Cycle} \Rightarrow \mathsf{Directed}$), in a feature model ϕ. The idea is simple: We check that the feature model implies the assumption ($\models \phi \Rightarrow \psi$). Phrased differently, we check whether $\phi \wedge \neg\psi$ is satisfiable; if it is, the feature model is incorrect as there exists a valid feature selection in ϕ that does not satisfy ψ.

In a practical setting, a domain expert can test a feature model by creating a list of assumptions that the feature model must satisfy. As in all testing, there is a certain redundancy in that we need to specify knowledge about features twice (in the feature model and in the assumptions) and then check that both align. As usual, testing can only show the presence of errors and not their absence.

[4] Typically, SAT solvers require formulas to be in conjunctive normal form, but these details should be hidden by feature-modeling tools.

Example 10.3 Here is a list of facts that could be used to test the feature model for the graph example:

$$\begin{aligned} \mathsf{Kruskal} &\Rightarrow \mathsf{Weighted} \\ \mathsf{Prim} &\Rightarrow \mathsf{Weighted} \\ \neg(\mathsf{Prim} &\wedge \mathsf{Kruskal}) \\ &\ldots \end{aligned}$$

The first two facts state that an `MST` algorithm requires `Weighted` graphs. The third states that both `Prim` and `Kruskal` algorithms will never both be present in a graph product, and so on. In our example, all tests pass. □

10.1.4 Dead Features and Mandatory Features

Next, we might want to know if a feature is dead or mandatory. A *dead feature* is never used in any product. In contrast, a *mandatory feature* is always used in every product.

Given ϕ of a feature model, there is at least one valid feature selection with feature `f`, iff $\phi \wedge \mathtt{f}$ is satisfiable, and there is at least one valid feature selection without feature `f`, iff $\phi \wedge \neg\mathtt{f}$ is satisfiable. A feature is *dead* if there is no valid feature selection with it ($\neg\mathrm{SAT}(\phi \wedge \mathtt{f})$) and *mandatory* if there is none without it ($\neg\mathrm{SAT}(\phi \wedge \neg\mathtt{f})$). To detect all dead (or mandatory) features, we simply iterate over all features.

Example 10.4 The feature model depicted in Fig. 10.2 has no dead features: GraphLibrary and EdgeType are mandatory. If we make also feature Undirected mandatory ($\phi'' = \phi \wedge \mathsf{Undirected}$), the features Directed and Cycle become dead features. In an inconsistent feature model, as ϕ' from Example 10.2, all features are simultaneously dead and mandatory (that is why we should rule out this fact first). □

A typical application of detecting dead features is to report warnings in the feature model editor. Also, an editor may issue a warning as *false optional feature* if analysis reports that a feature that is modeled as optional feature (or part of a choice or alternative group) is actually mandatory in all valid feature selections. Dead and false optional features can be considered code smells of feature models that indicate possible defects (see Chap. 8).

10.1.5 Constraint Propagation

As a user chooses features during feature selection, some features may no longer be selectable (they would invalidate the feature model) and others become required. A good editor can provide tool support to infer feature selections by automatically disabling or hiding unavailable features and selecting implied features automatically. This mechanism is called *constraint propagation*.

So far, we specified feature selections as a set of features and assumed that *all* features not within the set are deselected. In contrast, in a *partial feature selection*, we have not yet made a decision about all features, in particular, as product configuration and derivation is often an incremental process. Therefore, there are three possibilities: a feature is selected, a feature is deselected, or no decision has been made. As a consequence, we specify a partial feature selection with two sets: the set of selected features ($\mathtt{S} \subseteq \mathtt{F}$) and the set of deselected features ($\mathtt{D} \subseteq \mathtt{F}$, with $\mathtt{S} \cap \mathtt{D} = \emptyset$).

Determining which features must be selected or deactivated given a partial feature selection is similar to detecting dead or mandatory features. We encode a partial feature selection with the sets $\mathtt{S}$ and $\mathtt{D}$ as predicate $\mathtt{pfs(S,D)}$:

$$\mathtt{pfs(S,D)} = \bigwedge_{\mathtt{s} \in \mathtt{S}} \mathtt{s} \wedge \bigwedge_{\mathtt{d} \in \mathtt{D}} \neg \mathtt{d}$$

A partial feature selection is *valid*, iff $\phi \wedge \mathtt{pfs(S,D)}$ is satisfiable. We say a feature $\mathtt{f}$ is *deactivated* or no longer selectable, iff $\phi \wedge \mathtt{pfs(S,D)} \wedge \mathtt{f}$ is *not* satisfiable. Conversely, we say a feature is *activated* or must be selected, iff $\phi \wedge \mathtt{pfs(S,D)} \wedge \neg \mathtt{f}$ is *not* satisfiable.

Example 10.5 In Fig. 10.3 (p. 247), we showed a screenshot from FeatureIDE, where we selected feature Cycle. Due to the implication $\mathtt{Cycle} \Rightarrow \mathtt{Directed}$, the selection is propagated automatically to feature Directed. Technically, we see that for a partial selection $\mathtt{S} = \{\mathtt{Cycle}\}$, $\mathtt{D} = \{\}$, the corresponding formula is a contradiction:

$$\begin{aligned} &\phi \wedge \mathtt{pfs(D,S)} \wedge \neg \mathsf{Directed} \\ &= \ldots \wedge (\mathsf{Cycle} \Rightarrow \mathsf{Directed}) \wedge \mathsf{Cycle} \wedge \neg \mathsf{Directed} \end{aligned}$$

Since features Directed and Undirected are mutually exclusive, the selection is further propagated to deactivate feature Undirected. We derive this again, by determining that the corresponding formula is a contradiction:

$$\begin{aligned} &\phi \wedge \mathtt{pfs(S,D)} \wedge \mathsf{Undirected} \\ &= \ldots \wedge \neg(\mathsf{Directed} \wedge \mathsf{Undirected}) \wedge (\mathsf{Cycle} \Rightarrow \mathsf{Directed}) \wedge \\ &\quad \mathsf{Cycle} \wedge \mathsf{Undirected} \end{aligned}$$

In FeatureIDE, disabled and propagated selections are updated instantaneously during interactive editing. □

How to communicate the three possible states of whether a feature is selected, deselected, or yet open is a tricky user interface problem; one possibility is to use different symbols instead of normal check boxes as shown in Fig. 10.3 (right).

For an efficient mechanism to propagate constraints for a set of features with a minimal number of SAT-solver calls, see Janota's algorithm (Janota 2010).

10.1.6 Number of Valid Feature Selections

A question that managers ask is: How many valid feature selections does a feature model allow? Phrased differently: How many distinct products are part of this product line?

From a feature diagram without cross-tree constraints, a simple recursive algorithm calculates the number:

$$\begin{aligned}
&\mathit{count}\ \mathsf{root}(\mathtt{c}) &&= \mathit{count}(\mathtt{c})\\
&\mathit{count}\ \mathsf{mandatory}(\mathtt{c}) &&= \mathit{count}(\mathtt{c})\\
&\mathit{count}\ \mathsf{optional}(\mathtt{c}) &&= \mathit{count}(\mathtt{c}) + \mathtt{1}\\
&\mathit{count}\ \mathsf{and}(\mathtt{c_1}, \ldots, \mathtt{c_n}) &&= \mathit{count}(\mathtt{c_1}) * \ldots * \mathit{count}(\mathtt{c_n})\\
&\mathit{count}\ \mathsf{alternative}(\mathtt{c_1}, \ldots, \mathtt{c_n}) &&= \mathit{count}(\mathtt{c_1}) + \ldots + \mathit{count}(\mathtt{c_n})\\
&\mathit{count}\ \mathsf{or}(\mathtt{c_1}, \ldots, \mathtt{c_n}) &&= (\mathit{count}(\mathtt{c_1}) + \mathtt{1}) * \ldots * (\mathit{count}(\mathtt{c_n}) + \mathtt{1}) - \mathtt{1}\\
&\mathit{count}\ \mathsf{leaf} &&= \mathtt{1}
\end{aligned}$$

In a nutshell, function *count* is a recursive function that traverses the tree structure of a feature diagram from the root to the leaves. Depending on the type of feature, *count* is defined differently. This is implemented by pattern matching: for example, *count* optional(c) adds one to the number of valid feature selections and proceeds recursively with the subfeatures of the feature in question; *count* alternative($\mathtt{c_1}, \ldots, \mathtt{c_n}$) sums the valid feature selections of the alternative subfeatures of the feature in question. The recursion terminates when the features at the leaves of the tree are reached (*count* leaf).

Example 10.6 Ignoring the two cross-tree constraints, the simplified feature valid feature selections:

$$\begin{aligned}
\mathit{count}(\mathtt{f}) &= \mathtt{1}\ \text{//for all leaf nodes}\\
\mathit{count}(\mathsf{EdgeType}) &= \mathit{count}(\mathsf{Directed}) + \mathit{count}(\mathsf{Undirected}) = \mathtt{1} + \mathtt{1} = \mathtt{2}\\
\mathit{count}(\mathsf{MST}) &= \mathit{count}(\mathsf{Prim}) + \mathit{count}(\mathsf{Kruskal}) = \mathtt{1} + \mathtt{1} = \mathtt{2}\\
\mathit{count}(\mathsf{Algorithm}) &= (\mathit{count}(\mathsf{Cycle}) + \mathtt{1}) * (\mathit{count}(\mathsf{ShortestPath}) + \mathtt{1}) *\\
&\qquad (\mathit{count}(\mathsf{MST}) + \mathtt{1}) - \mathtt{1}\\
&= (\mathtt{1} + \mathtt{1}) * (\mathtt{1} + \mathtt{1}) * (\mathtt{2} + \mathtt{1}) - \mathtt{1} = \mathtt{11}
\end{aligned}$$

$$\begin{aligned} \mathit{count}(\textsf{GraphLibrary}) = &\ \mathit{count}(\textsf{Mandatory}(\textsf{EdgeType})) * \\ &\ \mathit{count}(\textsf{Optional}(\textsf{Weighted})) * \\ &\ \mathit{count}(\textsf{Optional}(\textsf{Algorithm})) \\ = &\ \texttt{2} * (\texttt{1} + \texttt{1}) * (\texttt{11} + \texttt{1}) = \texttt{48} \end{aligned}$$

□

For feature models with cross-tree constraints, the number is not easy to determine. A single cross-tree constraint can already eliminate a huge number of valid feature selections. For small feature models, we can simply count the solutions (for example, in a brute-force fashion, or with a SAT solver or binary decision diagrams). Fernandez-Amoros et al. (2009) have investigated a more sophisticated algorithm that can deal with certain kinds of cross-tree constraints.

Overall, this metric is of questionable utility. Due to the combinatorial number of feature selections in most product lines, a huge number is produced. Unless you like large numbers, saying that a product line yields 15 trillion valid feature selections in contrast to 3 quintillion of another, the number itself provides little insight. Most tools only provide approximations, such as an upper bound ignoring cross-tree constraints or an information on a small lower bound ("more than 1,000 valid feature selections"), which are cheap to compute and sufficient for many practical tasks.

10.1.7 Comparing Feature Models

Given two feature models ϕ_1 and ϕ_2, what is their relationship? Does ϕ_1 define the same set of feature selections (products) than ϕ_2 Is ϕ_1 a *generalization* of ϕ_2 (meaning that the set of products of ϕ_1 includes those of ϕ_2)? Or conversely, is ϕ_2 a *specialization* of ϕ_1?

These questions about the relationship between two feature models arose early in feature modeling. When a designer edits a feature model, she wants to know if her changes have altered the set of existing valid products. Enlarging the set of products may be acceptable, but eliminating products (particularly those that have been fielded) is often not. However, except for trivial cases such as adding or removing a single feature, the relationship is not always obvious. Even simple feature models are of sufficient complexity to make analyzing their relationships by manual inspection difficult.

Changing (improving) the structure of a feature model, while preserving all feature selections it describes is related to refactorings, a topic we considered in more depth in Chap. 8. A *feature-model refactoring* is an edit to a feature model that does not alter the set of legal feature selections (equivalent to a variability-preserving refactoring in Sect. 8.2.2).

In Fig. 10.4, we describe four possible relationships that we want to identify. A feature-model refactoring preserves exactly the set of valid feature selections, whereas a specialization removes feature selections without adding new ones and

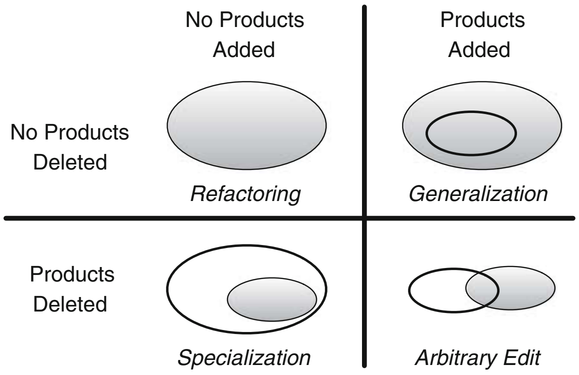


Fig. 10.4 Refactoring, specializations, and generalizations of feature models, between an old feature model (feature selections described by the *solid circle*) and a new feature model (feature selections described by *shaded area*)

a generalization adds valid feature selections without removing any. All differences that both add and remove valid feature selections are classified as arbitrary edits.

One approach to classify the relationship between two feature models is to describe their difference in terms of a set of known transformations (Alves et al. 2006). If we can express a feature-model difference in terms of a chain of well-known transformations, we can deduce the nature of the difference based on the properties of the transformations, for example, whether the difference represents a refactoring. In a feature-model editor, it would be possible to provide editing operations that are guaranteed to be refactorings.

A more general solution supporting arbitrary edits (without the limitations of a structured editor and more flexible with regard to cross-tree constraints) is again based on Boolean satisfiability. Two feature models ϕ_1 and ϕ_2 are equivalent when their propositional formulas are equivalent, that is, $\models \phi_1 \Leftrightarrow \phi_2$, or operationalized for a SAT solver $\neg\texttt{SAT}\big(\neg(\phi_1 \Leftrightarrow \phi_2)\big)$. Similarly, ϕ_1 is a *specialization* of ϕ_2 and ϕ_2 is a *generalization* of ϕ_1, iff $\models \phi_2 \Rightarrow \phi_1$. Thüm et al. (2009) discuss in more detail the different kinds of relationships and how to efficiently encode them as Boolean satisfiability problems, even for very large feature models.

Example 10.7 In Fig. 10.5, we show two feature models $\phi_{\texttt{right}}$ and $\phi_{\texttt{left}}$ ($\phi_{\texttt{left}}$ is an excerpt of the graph example). The propositional formulas for both the models are:

$$\begin{aligned}
\phi_{\texttt{left}} &= \mathsf{Algorithm} \wedge ((\mathsf{Cycle} \vee \mathsf{ShortestPath} \vee \mathsf{MST}) \Leftrightarrow \mathsf{Algorithm})\\
\phi_{\texttt{right}} &= \mathsf{Algorithm} \wedge (\mathsf{Cycle} \Rightarrow \mathsf{Algorithm}) \wedge (\mathsf{ShortestPath} \Rightarrow \mathsf{Algorithm})\\
&\quad \wedge (\mathsf{MST} \Rightarrow \mathsf{Algorithm}) \wedge (\mathsf{Cycle} \vee \mathsf{ShortestPath} \vee \mathsf{MST})
\end{aligned}$$

A SAT solver can prove $\phi_{\texttt{right}} \Leftrightarrow \phi_{\texttt{left}}$. □

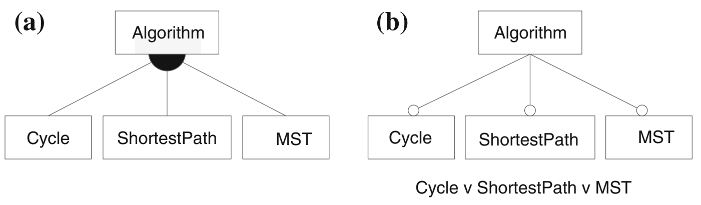


Fig. 10.5 Two equivalent feature models

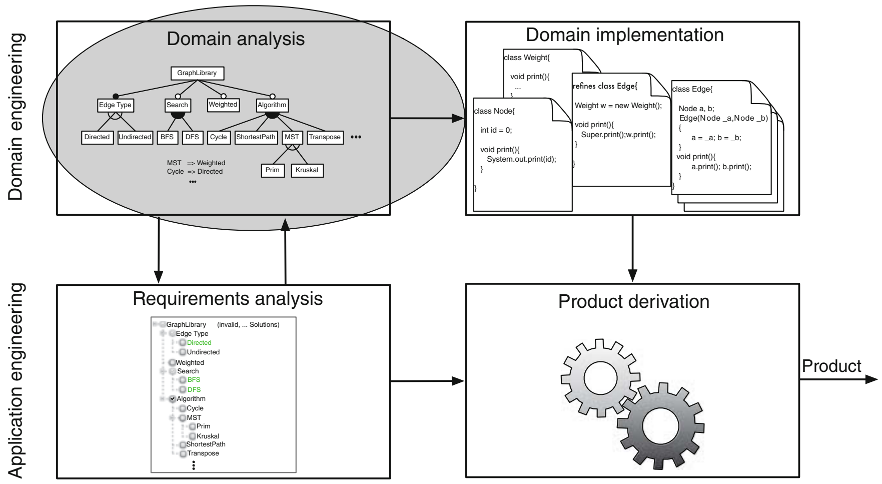
A typical use case for the comparison of feature models is during feature-model editing. For example, FeatureIDE displays after each edit whether all changes since the feature model was last saved forms a refactoring, a specialization, or a generalization. It also lists examples of added or removed feature selections. Furthermore, the analysis can be used to compare and possibly merge two independently changed feature models (Thüm et al. 2009).

10.1.8 Other Feature-Model Analyses

Other analyses detect redundancies, explain feature selections, optimize selections, and calculate various metrics. Other variability models (with cardinalities, attributes and non-Boolean features) have been explored along with different kinds of solvers, from SAT solvers as in this chapter, to binary decision diagrams, to solvers for constraint satisfaction problems. For instance, an interesting class of problems deals with attributes to features that describe nonfunctional properties, such as costs, memory consumption, performance impact, footprint, and security; optimization algorithms can then find the best solution (for some target function or some nonfunctional constraints) given a partial feature selection (Benavides et al. 2005; Sincero et al. 2010; Siegmund et al. 2011). For an introduction and overview of the state of the art on feature-model analysis and solvers, see the survey by Benavides et al. (2010).

10.2 Analysis of Feature-to-Code Mappings

We reviewed in the last section analyses that focus on the problem space with feature model and feature selections. Now, we investigate the *mapping from features to code*, that is, the mapping from problem space to solution space, as shown in Fig. 10.6. By leveraging the analyses of the previous section, we show how to detect unused modules in feature-oriented programming, dead code in preprocessor-based implementations, and elaborate on issues regarding build systems that can complicate analyses (see Sect. 5.2 p. 105). We will not yet look at structures in the source code such as methods or statements (we will do that in Sect. 10.3); instead, we consider code fragments as arbitrary text sequences. Specifically, we explore the following:



```
class Weight{

  void print(){
    ...
  }
```

```
class Node{

  int id = 0;

  void print(){
    System.out.print(id);
  }

}
```

```
refines class Edge{

 Weight w = new Weight();

 void print(){
   Super.print();w.print();
 }

}
```

```
class Edge{

  Node a, b;
  Edge(Node _a,Node _b)
  {
      a = _a; b = _b;
  }
 void print(){
      a.print(); b.print();
  }
}
```

Fig. 10.6 Analysis of feature-to-code mappings incorporates knowledge about the mapping and the feature model, but not yet about structures in the source code

```
1 line 1
2 #ifdef A
3 line 3
4 #ifdef B
5 line 5
6 #endif
7 #else
8 line 8
9 #endif
```

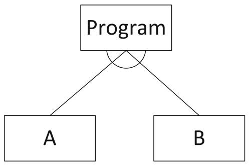


Fig. 10.7 Simple example of a dead code fragment in Line 5

- Which code fragments are never included in any product?
- Which code fragments are included in all products?
- Which features have no influence on the product portfolio?

10.2.1 Dead Code

Our first goal is to find *dead code*—fragments that are never included in any valid feature selection. Dead code can be an indicator for an incorrect mapping or an over-constrained feature model. Look at Fig. 10.7: Line 5 is included only if the features `A` and `B` are both selected, but the feature model specifies both features as mutually exclusive. That is, Line 5 can be never included in *any* product of the product line. Developers were likely unaware that features `A` and `B` are defined as mutually exclusive, or the feature model was too strict and thus both features could be optional.

Detecting dead code is different from traditional detection of unreachable code. Compilers analyze a program's control flow of a *single product* using static analyses. In contrast, we find code that is dead with regard to feature selections in a product line. Performing control-flow analysis on a whole product line requires more sophisticated techniques outlined in Sect. 10.3.7.

To identify dead code, we need to reason about the mapping from features to code. A code fragment can be a plug-in of framework (Sect. 4.3, p. 79), a file that is conditionally excluded by a build system (Sect. 5.2, p. 105), a block of code guarded by conditional-compilation directives as in Fig. 10.7 (Sect. 5.3, p. 110), a feature module (Sect. 6.1, p. 130), an aspect (Sect. 6.2, p. 141), or some other variable code. We can even regard a parameter in a build script (for example, calling the compiler at different optimization levels) or flags generated in a configuration file (Sect. 4.1, p. 66) as analyzable code fragments.

Formally, we describe the mapping as a function from code fragments (from the set `C` of all code fragments) to sets of products represented by feature selections ($\texttt{pc: C} \rightarrow 2^{2^{\texttt{F}}}$). That is, we map each code fragment to the set of products in which it is included. As a compact representation of large sets of feature selections and corresponding products, we use a *presence condition*—a propositional formula representing a set of feature selections. This is in line with the propositional formula representing the set of valid feature selections in a feature model (see Sect. 2.3.3 p. 31). A presence condition defines the feature selections in which a code fragment is present. For example, in Fig. 10.7, the code fragment in Line 3 has the presence condition `A` and is included in all products with feature `A`; Line 3 has the presence condition $\texttt{A} \wedge \texttt{B}$ and is included in all products with the features `A` and `B`; and Line 8 has presence condition $\neg\texttt{A}$ and is included in all products that do not include feature `A`. In this section, we use function `pc(c)` to denote the presence condition of a code fragment `c` in the form of a propositional formula.

A code fragment is *dead* if it is never included in any product of a product line. Since the representation of the feature model as propositional formula ϕ represents all valid feature selections (products), a code fragment `c` is dead iff the conjunction of presence condition and feature model is not satisfiable: $\neg\texttt{SAT}(\phi \wedge \texttt{pc(c)})$. That is, there is no feature selection that is both valid according to a feature model and that fulfills the presence condition.

Example 10.8 Returning to our example of Fig. 10.7, we have the presence conditions $\top$, `A`, $\texttt{A} \wedge \texttt{B}$, and $\neg\texttt{A}$ for the Lines 1, 3, 5, and 8 respectively. The predicate of the feature model is $\phi = \texttt{root} \wedge (\texttt{A} \vee \texttt{B}) \wedge \neg(\texttt{A} \wedge \texttt{B})$. Line 5 is dead because $\phi \wedge \texttt{A} \wedge \texttt{B}$ is unsatisfiable. □

In a similar manner, we can detect code fragments that are included in all products. Such code fragments are *mandatory*. A code fragment `c` is mandatory iff $\models \phi \Rightarrow \texttt{pc(c)}$, that is, $\neg\texttt{SAT}(\phi \wedge \neg\texttt{pc(c)})$.

10.2.2 Abstract Features

As introduced in Sect. 2.3.5, *abstract features* are used in some notations of feature models, but are not mapped to any code. That is, selecting or not selecting abstract features does not have an influence on product derivation, and those features may be skipped during the requirements analysis process.

As a conservative approximation, every feature that does not occur syntactically in any presence condition is abstract. This approximation is usually sufficient in practice, but would not catch corner cases such as a presence condition $\mathtt{A} \vee \neg\mathtt{A}$, which contains the feature name but does not influence product derivation. For a more precise analysis, we can again encode the analysis as a corresponding Boolean satisfiability problem (Thüm et al. 2011a): Feature f is abstract, iff the following formula is satisfiable:

$$\bigvee_{\mathtt{c}\in\mathtt{C}} \mathtt{pc(c)}[\mathtt{f} \rightarrow \top] \oplus \mathtt{pc(c)}[\mathtt{f} \rightarrow \bot]$$

where $\mathtt{p}[\mathtt{A} \rightarrow \mathtt{B}]$ denotes substituting all occurrences of A by B in predicate p and $\oplus$ denotes exclusive or. In our example $\mathtt{A} \vee \neg\mathtt{A}$, we would substitute A both by true ($\top$) and false ($\bot$) to yield $(\top \vee \neg\top) \oplus (\bot \vee \neg\bot)$, which is true; so we know the Boolean value of feature A has no impact on selecting code fragments.

10.2.3 Determining Presence Conditions

Previously, we assumed that we knew the presence conditions for all code fragments. In Example 10.8, we used a simple presence condition without explanation. Although, we may separately model presence conditions of implementation artifacts (Metzger et al. 2007), we argue that it is usually more convenient and reliable to extract them directly from the variability in the implementation. In this section, we discuss how to extract presence conditions for different implementation mechanisms.

Feature-Oriented Programming

In Sect. 6.1, we considered feature modules as code fragments to analyze. Developers mostly use an implicit one-to-one mapping between features and feature modules, that is, the feature module has the same name as the feature in the feature model. Therefore, we can extract a mapping as follows: A feature module X has the presence condition $\mathtt{pc(X)} = \mathtt{X}$ (referring to a feature with the same name). Hence, determining presence conditions is trivial.

Of course also explicit external mappings are possible, for example, a table describing the presence condition for every module or a build system selecting which modules to compose. Especially with regard to extra modules for feature interactions (Sect. 9.4.7, p. 230) presence conditions such as $A \wedge B$ are common.

Conditional Compilation with the C Preprocessor

Extracting a presence condition for code fragments using conditional compilation (Sect. 5.3, p. 110) is also straightforward. In the context of the C preprocessor, a code fragment refers to a sequence of code lines within a file; code fragments are separated by conditional-compilation directives. Macros that control conditional compilation are often mapped directly to features or have a simple mapping (for example, in the Linux kernel, macro 'CONFIG_X' represents feature X).

As described in Sect. 5.3, the C preprocessor has the directives #ifdef, #ifndef, #if, #elif, #else, and #endif, which can be nested. Instead of explaining in detail how to extract the mapping, we simply give the example in Fig. 10.8. For a precise description, see the formalization by Tartler et al. (2011).

Determining a presence condition for code that includes conditional-compilation directives is not always as simple as in Sect. 5.3.2. Using #define and #undef directives, developers can activate and deactivate macros within the source code during the execution of the preprocessor (possibly depending on other features using conditional compilation on macro definitions). A precise analysis is outside the scope of our book and discussed elsewhere (Hu et al. 2000; Favre 2003; Latendresse 2003, 2004; Kästner et al. 2011; Tartler et al. 2011; Gazzillo and Grimm 2012). However, simple, disciplined preprocessors (or preprocessor usage) can significantly ease analyses. For example, we recommend not changing the definition of macros that denote features within the source code; so the intuitive extraction procedure above can be used.

```
 1 #ifdef CONFIG_A
 2 line 2
 3 #elif defined(CONFIG_B)
 4 line 4
 5 #else
 6 line 6
 7 #endif
 8 line 8
 9 #if defined(CONFIG_X) || (defined(CONFIG_Y) &&
          defined(CONFIG_Z))
10 line 10
11 #ifdef CONFIG_A
12 line 12
13 #ifndef CONFIG_B
14 line 14
15 #else
16 line 16
17 #endif
18 #endif
19 #endif
```

Line	Presence condition
2	A
4	$B \wedge \neg A$
6	$\neg B \wedge \neg A$
8	$\top$
10	$X \vee (Y \wedge Z)$
12	$(X \vee (Y \wedge Z)) \wedge A$
14	$(X \vee (Y \wedge Z)) \wedge A \wedge \neg B$
16	$(X \vee (Y \wedge Z)) \wedge A \wedge B$

Fig. 10.8 Examples of presence conditions extracted from conditional compilation

More modern annotation-based implementation strategies enforce a direct mapping. For example, FeatureMapper (Sect. 5.3.3, p. 113) and virtual separation of concerns (Sect. 7.4, p. 184) store feature-code mappings separately, for example, in a table explicitly mapping code fragments to presence conditions. Here, extracting presence conditions is trivial.

Build Systems

A build system selects which files to compile and how (Sect. 5.2). As build systems control the inclusion of entire files or directories, code fragments in this context can refer to plug-ins (see Sect. 4.3, p. 79), aspects (see Sect. 6.2, p. 141), feature modules, or any other files or containers. As build systems also control how (for example, with which parameters) files are compiled and initiate generators, we can also determine presence conditions for compiler parameters or settings in configuration files.

In the simplest case, a build system maintains a list of presence conditions for each file. Unfortunately, determining presence conditions from a build system is not always easy, because most build systems are written in sophisticated Turing-complete scripting languages. Extracting presence conditions is often undecidable, because many build systems may perform arbitrary computations by calling shell scripts.

Analysts wanting to extract variability from build systems can pursue different strategies. First, they can use a disciplined build system with limited expressiveness designed for analysis (for example, a system providing a direct mapping between presence conditions and files). Second, automated tools can try to detect common patterns used in existing build scripts, however, accuracy will depend on whether and how those patterns are used (for example, Berger et al. (2010a) and Nadi and Holt (2012) describe experience with such extraction for the Linux kernel). Finally, an analyst could perform different kinds of more heavyweight dynamic and static analysis on the build script, such as symbolic execution (Tamrawi et al. 2012).

There is a trade-off between how expressive and how analyzable the build system is. It may be that the expressiveness provided by contemporary build systems is not needed, but is used simply because the developers are familiar with it. The more expressive the build system's language, the less accurate and the more difficult the analysis process becomes. Imprecision in the analysis can yield to both false positives and false negatives when searching for dead code fragments and abstract features. For many purposes, restricted domain-specific languages would suffice and allow precise analysis, though some migration effort may be necessary in existing projects.

In practice, it is typical to combine conditions extracted from the build system with conditions from other implementation mechanisms, such as preprocessors.

Parameters

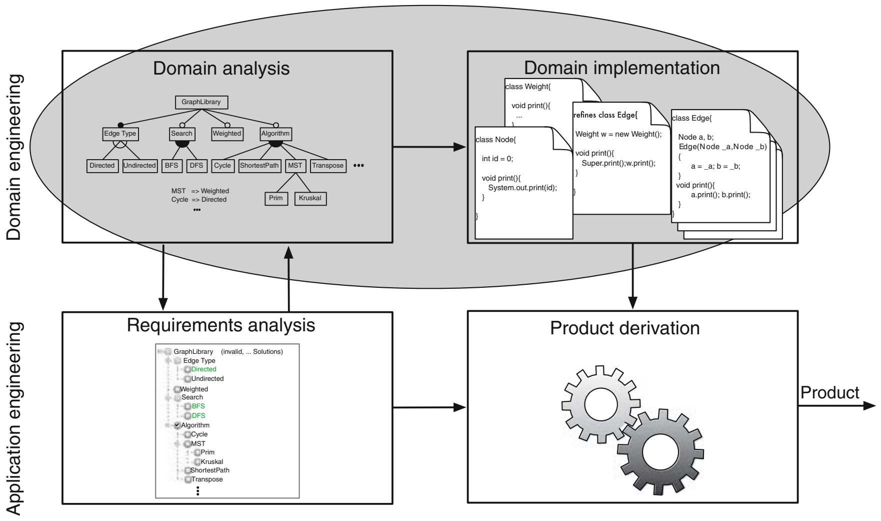
For software product lines that use run-time variability (see Sect. 4.1, p. 66), static presence conditions are the most difficult to extract. With intra-procedural control-flow and data-flow analysis, we could attempt to trace configuration parameters to specific code fragments. Detecting feature code in a product line implemented with run-time parameters is conceptually similar to detecting unreachable code in compilers (with some extra knowledge about these parameters). However, as parameters can be passed throughout the program and assigned and modified, we would need sophisticated and computationally expensive abstract interpretation or slicing analysis. Further, such analysis is always incomplete or unsound, so either false positives or false negatives cannot be avoided (see Rice's theorem).[5]

When parameters are used in a restricted and disciplined fashion, specific analysis techniques could in principle detect many presence conditions (Haase 2012; Ouellet et al. 2012). Anyway, implementations based on compile-time variability (especially advanced language-based and tool-based approaches) are naturally easier to analyze statically than approaches based on run-time variability. To perform variability-aware analysis, a reliable extraction of presence conditions is important and disciplined implementation approaches can simplify that task significantly.

10.3 Analysis of Domain Implementations

After analyzing feature models and the mapping from features to code, we now focus on analyzing variability in program structures, such as function calls or statements. We call these analyses *variability-aware analysis*, because they perform traditional analyses, such as type checking and model checking, but they incorporate knowledge about variability in the system. Again, we want to analyze and ensure properties for all possible products of a product line. These analyses build on top of the analyses we have presented earlier, and reason about all kinds of domain-engineering artifacts (feature models, domain implementations, and the mapping):

[5] Rice's theorem says in the general result no static analysis can prove a non-trivial property for any programs in a finite time. Of course, by restricting the domain of programs, by requiring certain structures or properties of these programs, a non-trivial property can be proven. This is analogous to the Halting Problem—for straight-line programs, the Halting Problem is solvable.



```
class Weight{

  void print(){
    ...
  }
```

```
class Node{

  int id = 0;

  void print(){
    System.out.print(id);
  }

}
```

```
refines class Edge{

 Weight w = new Weight();

 void print(){
   Super.print();w.print();
 }

}
```

```
class Edge{

  Node a, b;
  Edge(Node _a,Node _b)
  {
       a = _a; b = _b;
  }
 void print(){
       a.print(); b.print();
  }
}
```

The idea of variability-aware analysis is not to invent new kinds of analysis techniques, but to *lift* existing analysis techniques developed for individual programs to entire product lines (that is, to domain artifacts). Examples of established analyses that we want to lift include type checking, model checking, data-flow analysis, and deductive verification. The goal is to perform the same analysis on a product line that we could perform on every possible product separately. Ideally, variability-aware analysis should yield the same results, but in a more efficient way.

Let us explain the vision of variability-aware analysis in Fig. 10.9. We have two possible paths to check a property for an entire product line:

- **Brute-force analysis**. Starting from a product line, we can derive a product per valid feature selection (Step 1). For each product, we can now perform a given off-the-shelf analysis (Step 2). For example, we could compile the source code to detect syntax errors and type errors. If we repeat that process for every valid product (exponentially many, in the worst case), we can aggregate the results and determine whether the property holds for all products of the product line (Step 4).
- **Variability-aware analysis**. We analyze the domain artifacts of the product line, without checking all products of a product line individually. Variability-aware analysis produces a result for the entire product line (for example, 'the property holds for all products', or 'the property does not hold for products with feature A'). From the result, we can derive whether the property holds for a specific product (Step 4).

Given an ideal variability-aware analysis, both paths should come to the same conclusions. That is, variability-aware analysis should yield the same result as applying an existing analysis in a brute-force approach. At the same time, we expect that variability-aware analysis is typically much faster, because it can exploit similarities and reuse analysis results across products. For example, if multiple products share

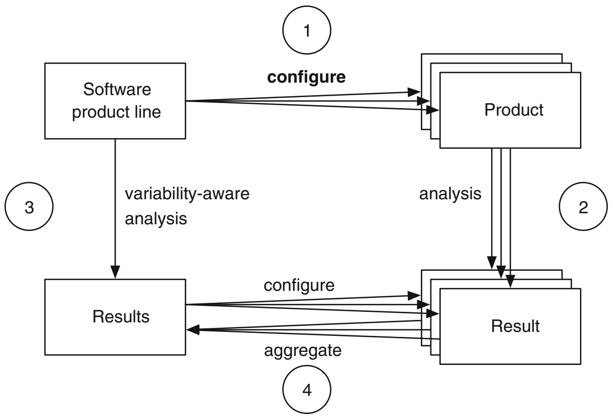


Fig. 10.9 Ideally, variability-aware analysis should reach the same result as traditional analysis applied to all products in isolation

code (from the base code or from some feature code), variability-aware analysis might only need to analyze it once and not over and over again for each possible feature selection. (Note: Not all variability-aware analyses follow this ideal picture. Some provide approximations that are not exact, but still useful and much faster to compute).

We illustrate variability-aware analysis with type checking, because it is well-understood and comparably easy to explain. Type checking of product lines is only interesting for implementation approaches with compile-time variability though. In approaches with run-time variability, such as parameters (see Sect. 4.1, p. 66), only a single program is compiled (and checked), which can be both a strength (easy check for type errors) and a weakness (some errors caught only at run time). We use examples from type checking product-line implementations based on preprocessors and feature-oriented programming.

10.3.1 Design Space

There is a large design space of different analyses and researchers have explored many different strategies. Before we discuss specific analyses, we introduce some additional terminology that helps to distinguish different variability-aware analyses within the design space.

First, different kinds of analysis check different *properties* and give different guarantees. For example, a type system checks well-typedness of programs to ensure the absence of a certain class of errors, whereas model checking verifies that the behavior of a program satisfies a given specification.

Second, in a product-line context, the issue arises of how to specify the expected behavior of a product line. Can we specify the behavior of each feature in isolation, should we provide a specification per product, or is a single global specification for all products sufficient? For simplicity, here we always expect a global specification that must hold for all products, such as 'all products shall be well-typed' or 'there shall be no null-pointer exception during the execution of any product.'

Finally, there are different strategies to lift analyses to handle the large configuration space of a product line. We say a variability-aware analysis is *complete*, if it finds the same property violations that the brute-force approach would find (see Fig. 10.9). We say a variability-aware analysis is *sound*, if every property violation found in domain artifacts is also a property violation in a corresponding derived product.

10.3.2 Sampling Strategies

A first strategy, which is easy to apply, is to check only a (suitable) subset of all products of a product line with an off-the-shelf, single-product analysis. For example, we can choose interesting feature selections, derive the corresponding products, and simply compile them to find type errors in the sampled products. This corresponds taking the path 1–3 in Fig. 10.9 multiple times (though not in a brute-force fashion for all products).

The main question is how to select the sample of feature selections to analyze? Typically, we want to check only a small number of products, but achieve a high coverage according to some criterion. Among many others, possible coverage criteria in product lines are:

- *Feature coverage:* Select products such that every feature (from the problem space) is included in at least one product.
- *Feature-code coverage:* Select products such that every code fragment (from the solution space) is included in at least one product.
- *Pair-wise feature coverage:* Select products such that each pair of features is included in at least one product. Additionally, we can demand that for each feature pair (f, g) there is a product with f but without g and a product with g but without f, in addition to a product with both f and g.
- *N-wise feature coverage:* Much like pair-wise feature coverage, but all possible n-tuples of features should be included in at least one product.
- *Popular products and features:* Select products frequently used by customers or products with features that are often requested.
- *Domain-specific:* In many domains experts can provide suitable coverage criteria for the domain, for example, critical features such as transaction management in a database system.

Sampling with feature coverage may result in a poor detection error rate, because problems related to interactions between multiple features might not be detected (see Chap. 9). Pair-wise coverage attempts to address this problem by analyzing every pair of features, so that we can detect all interactions between two features. To achieve pair-wise coverage, only a moderate number of products are necessary: For example, Oster et al. (2010) shows an example of a feature model with 88 features that can be covered by 40 products and a feature model with 287 features that can be covered with 62 products. Also, n-wise coverage with larger n is possible; we may detect more interactions, but this approach requires much larger samples. When selecting a sample, we always have to face a trade-off between the number of products selected (analysis effort) and the desired coverage.

There are many other coverage criteria and combinations of them. Furthermore, there are different strategies to find the smallest (or a small) number of product that fulfills one or more coverage criteria, some of which require sophisticated analysis with SAT solvers that are outside our discussion (see Sect. 10.6, p. 277).

Note that sampling strategies are sound but always incomplete. Since we do not look at all products, we might miss errors. We cannot establish guarantees for an entire product line. However, when we find an error, we are sure that it actually is an error. In this respect, variability-aware analysis with sampling is similar to software testing and borrows from a large amount of research on combinatorial testing and test coverage.

10.3.3 Family-Based Type Checking of Preprocessor-Based Implementations

Next, we look at an example of how to analyze entire product lines: *family-based* type checking. We first illustrate family-based type checking of preprocessor-based implementations, and subsequently demonstrate its generality by applying it also to feature-oriented programming.

To illustrate family-based, variability-aware type checking, we slightly extend our graph example again, and use an almost trivial excerpt. As shown in Fig. 10.10, we extend the graph example from Fig. 5.9 (p. 112), such that nodes can optionally have a name, in addition to their id (for example, to store names if nodes represent persons). Feature Name introduces a new method getName, and, to provide the same interface without names, feature NoName provides the same method, but with a dummy implementation.

Together with the already known feature Color, our example has three features, which can be combined to eight different products. Obviously, for some of these products, a Java compiler will issue type errors: Selecting neither Name nor NoName leads to a dangling method invocation in the parameter of the print statement (Line 20; getWeight has not been declared); selecting both Name and NoName leads to a method declared twice.

To detect these kinds of errors with a brute-force approach, we have to derive and compile all eight products individually. While a brute-force approach seems acceptable for this example, it clearly does not scale for product lines with more features, as the number of products to check grows exponentially. Instead, we lift Java's type system to take variability into account.

```
 1 class Node {
 2     int id = 0;
 3
 4     //#ifdef NAME
 5     private String name;
 6     String getName() { return name; }
 7     //#endif
 8     //#ifdef NONAME
 9     String getName() { return String.valueOf(id); }
10     //#endif
11
12     //#ifdef COLOR
13     Color color = new Color();
14     //#endif
15
16     void print() {
17         //#if defined(COLOR) && defined(NAME)
18         Color.setDisplayColor(color);
19         //#endif
20         System.out.print(getName());
21     }
22 }
23 //#ifdef COLOR
24 class Color {
25     static void setDisplayColor(Color c){/*...*/}
26 }
27 //#endif
```

Fig. 10.10 Extended graph example implementing colors and optional names of nodes using preprocessor directives

Presence Conditions on Structures

Variability-aware type systems reasons about *presence conditions* in the source code. However, in contrast to the presence conditions for arbitrary textual *fragments* in Sect. 10.2, we now reason about presence conditions for structural program elements of in the domain implementation, such as variables, fields, and methods. In our example, the two methods getWeight have the presence condition Name (Line 6) and NoName (Line 9), respectively; the first statement in the main function has presence condition Color $\wedge$ Name (Line 18). We emphasize the mapping of presence condition to program elements (instead of lines of plain text) by showing an abstract-syntax tree of the code fragment that includes nodes for variability in Fig. 10.11 ($<$optional$>$ nodes denote optional subtrees with a presence condition). Again, we denote presence conditions with function `pc`. Depending on the implementation mechanism, extracting such mapping is more or less complex (usually straightforward for composition-based and *disciplined* annotation-based implementations; more difficult for undisciplined annotations; see also Sect. 5.3.4).

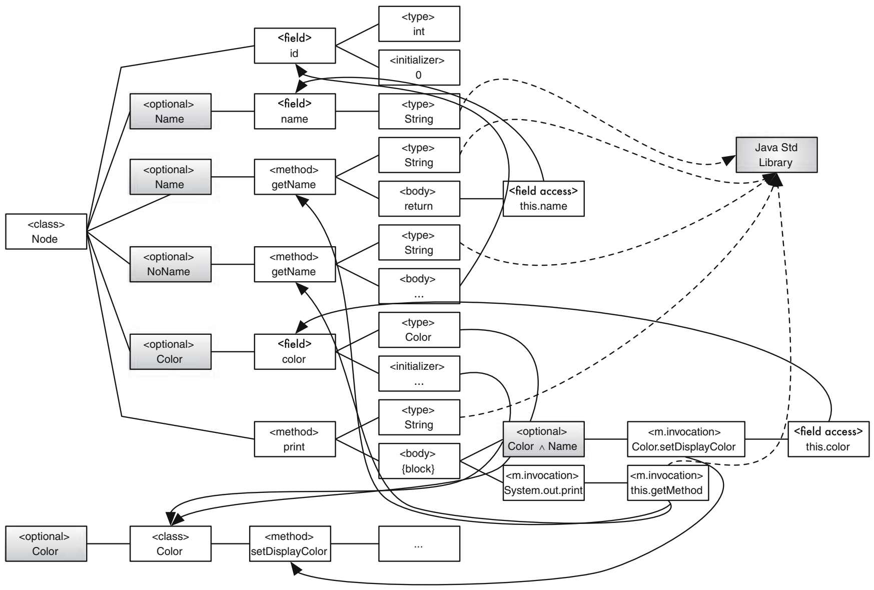
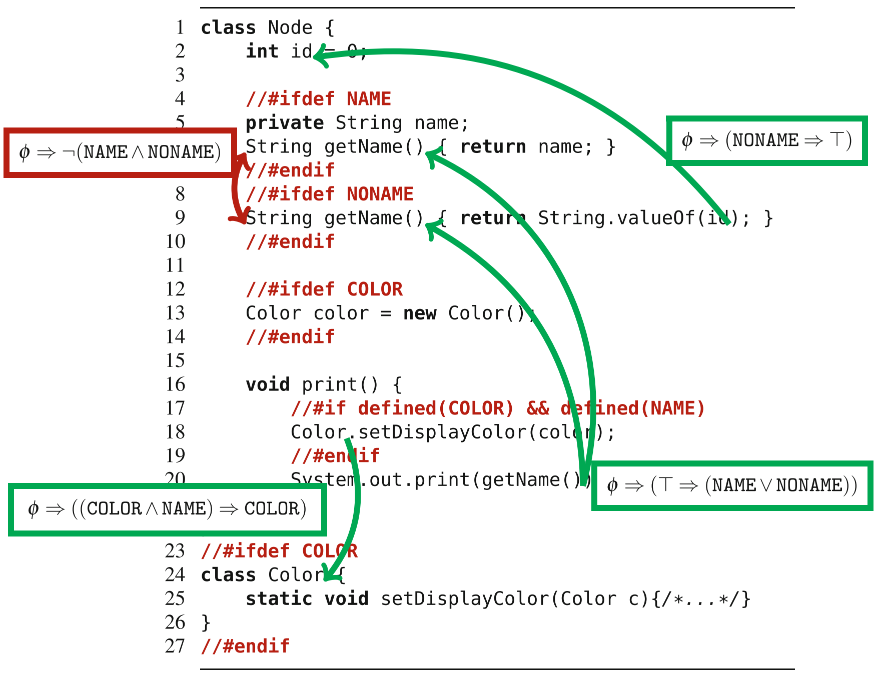


Fig. 10.11 Abstract syntax tree of the domain implementation of the graph example of Fig. 10.10, describing all variations

Reachability Constraints

A family-based analysis operates on a program representation that is variable. In our example, family-based type checking takes place at a variable abstract syntax tree that represents the whole space of possible products. Let us generalize from an example: When resolving a method invocation, there can be different target declarations in different products. The type system must ensure that all derivable products that contain the method invocation must also contain a corresponding method declaration as target (with an expected type). In our example, method getName is invoked in all products with presence condition `true` (Line 20, expected to return type String), but a corresponding method declaration is only present in products with the features Name or NoName (Lines 6 and 9, both returning type String). Just by comparing presence conditions within the product-line implementation, we can identify that products without feature Name and without feature NoName will contain a type error. If such feature selections are valid according to the feature model, we can issue an error message: "cannot resolve method getName() in Line 20 if ¬Name ∧ ¬NoName", as shown in Fig. 10.12.

A type system performs many other lookups, of fields, local variables, methods, classes, types, and so on. In all cases, we need to ensure that a target element is present whenever the source element is present (and often with additional constraints on types). For instance, a field can only have type Color and we can only instantiate class Color when a corresponding class declaration is present. More generally,


$\phi \Rightarrow \neg(\texttt{NAME} \wedge \texttt{NONAME})$
$\phi \Rightarrow (\texttt{NONAME} \Rightarrow \top)$
$\phi \Rightarrow (\top \Rightarrow (\texttt{NAME} \vee \texttt{NONAME}))$
$\phi \Rightarrow ((\texttt{COLOR} \wedge \texttt{NAME}) \Rightarrow \texttt{COLOR})$

```
 1  class Node {
 2      int id = 0;
 3
 4      //#ifdef NAME
 5      private String name;
        String getName() { return name; }
        //#endif
 8      //#ifdef NONAME
 9      String getName() { return String.valueOf(id); }
10      //#endif
11
12      //#ifdef COLOR
13      Color color = new Color();
14      //#endif
15
16      void print() {
17          //#if defined(COLOR) && defined(NAME)
18          Color.setDisplayColor(color);
19          //#endif
20          System.out.print(getName());

23  //#ifdef COLOR
24  class Color {
25      static void setDisplayColor(Color c){/*...*/}
26  }
27  //#endif
```

```
Found 2 type errors:
 - [NAME & NONAME] file Node.java:9
        'getName()' is already defined in 'Node'
 - [!NAME & !NONAME] file Node.java:20
        cannot resolve method 'getName()'
```

Fig. 10.12 Selected constraints in the graph example and corresponding output of a family-based type system

given a feature model ϕ, presence conditions $\texttt{pc}$, a source element $\texttt{s}$, and a set of target elements $\texttt{T}$, we can formulate the following generic constraint, which we call *reachability condition*:

$$\phi \Rightarrow \Big(\texttt{pc(s)} \Rightarrow \bigvee_{\texttt{t}\in\texttt{T}} \texttt{pc(t)}\Big)$$

If that constraint is not a tautology (that is, if its negation is satisfiable), we report an error message, indicating that there are products in the product line that do not compile. Once again, we use a SAT solver to perform this analysis. We can even pinpoint the error message to a set of feature selections by negating the constraint; for debugging, a SAT solver can provide specific feature selections to reproduce the error with an existing single-product analysis.

In a similar way, we can also detect redeclaration (or multiple-declaration) errors. In our example, we must not declare method getName twice. To this end, we check

Table 10.1 Reachability constraints in the graph example

Construct	Source	Target	Constraint
String (type reference)	5	JSL	$\phi \Rightarrow (\mathsf{Name} \Rightarrow \top)$
String (type reference)	6	JSL	$\phi \Rightarrow (\mathsf{Name} \Rightarrow \top)$
name (field access)	6	5	$\phi \Rightarrow (\mathsf{Name} \Rightarrow \mathsf{Name})$
String (type reference)	9	JSL	$\phi \Rightarrow (\mathsf{NoName} \Rightarrow \top)$
String.valueOf (method invocation)	9	JSL	$\phi \Rightarrow (\mathsf{NoName} \Rightarrow \top)$
id (field access)	9	2	$\phi \Rightarrow (\mathsf{NoName} \Rightarrow \top)$
Color (type reference)	13	24	$\phi \Rightarrow (\mathsf{Color} \Rightarrow \mathsf{Color})$
Color (instantiation)	13	24	$\phi \Rightarrow (\mathsf{Color} \Rightarrow \mathsf{Color})$
Color.setDisplayColor (method inv.)	18	25	$\phi \Rightarrow ((\mathsf{Color} \wedge \mathsf{Name}) \Rightarrow \mathsf{Color})$
color (field access)	18	13	$\phi \Rightarrow ((\mathsf{Color} \wedge \mathsf{Name}) \Rightarrow \mathsf{Color})$
System.out (field access)	20	JSL	$\phi \Rightarrow (\top \Rightarrow \top)$
PrintStream.print (method invocation)	20	JSL	$\phi \Rightarrow (\top \Rightarrow \top)$
getName (method invocation)	20	6, 9	$\phi \Rightarrow (\top \Rightarrow (\mathsf{Name} \vee \mathsf{NoName}))$
Color (type reference)	25	24	$\phi \Rightarrow (\mathsf{Color} \Rightarrow \mathsf{Color})$
getName (method redeclaration)	9	6	$\phi \Rightarrow \neg(\mathsf{Name} \wedge \mathsf{NoName})$

Source and target refer to lines in Fig. 10.10; JSL represents targets in the Java Standard Library with presence condition $\top$

that all declarations in a set of potentially conflicting declarations D are pair-wise mutually exclusive (within feature selections specified as valid by the feature model). We use the following constraint and report an error if it is not a tautology:

$$\phi \Rightarrow \bigwedge_{\mathtt{d_1 \in D,\ d_2 \in D,\ d_1 \neq d_2}} \neg\big(\mathtt{pc(d_1)} \wedge \mathtt{pc(d_2)}\big)$$

Example 10.9 We illustrate selected constraints derived from our graph example in Fig. 10.12. We give a full list of constraints in Table 10.1. Note that some references such as String and System refer to the Java Standard Library which is included in all products.
By solving the constraints, we can see that, without additional restrictions from a feature model, two constraints are violated. We can report corresponding error messages, as shown in Fig. 10.12 (bottom). More compactly, we could report the result of our analysis as "if (Name ⊕ NoName) then *well-typed* else *ill-typed*". The result is equivalent to the result gained from a brute-force application using the standard Java type system.
When the feature model is repaired, namely that Name and NoName are alternative features ($\phi \Rightarrow$ Name ⊕ NoName), all constraints we had to check in our example above are tautologies, so we now know that every valid product of our product line is well-typed. □

Performance

So, how does variability-aware type checking with a family-based strategy improve over the brute-force approach? Instead of checking reachability and redeclaration errors again and again in the generated code separately for each product, we formulate constraints over the space of all products. The important benefit of this approach is that we check variability locally in domain artifacts, where it occurs. For code that is not variable, we perform only a single check overall, instead of a check per product. For example, we check whether method System.out.print exists only once (instead of eight times for each product in the brute-force approach), and we check only two possible targets of the method invocation of getName, independent of whether feature Color is selected.

Rather than checking the surface complexity of up to 2^n products in isolation, family-based strategies analyze the domain artifacts of the entire product line and check only essential complexity where variability actually matters. Worst-case effort is still exponential, since developers could write product lines without any code sharing, but experience suggests that this happens rarely, because reuse is a key goal of product-line development.[6]

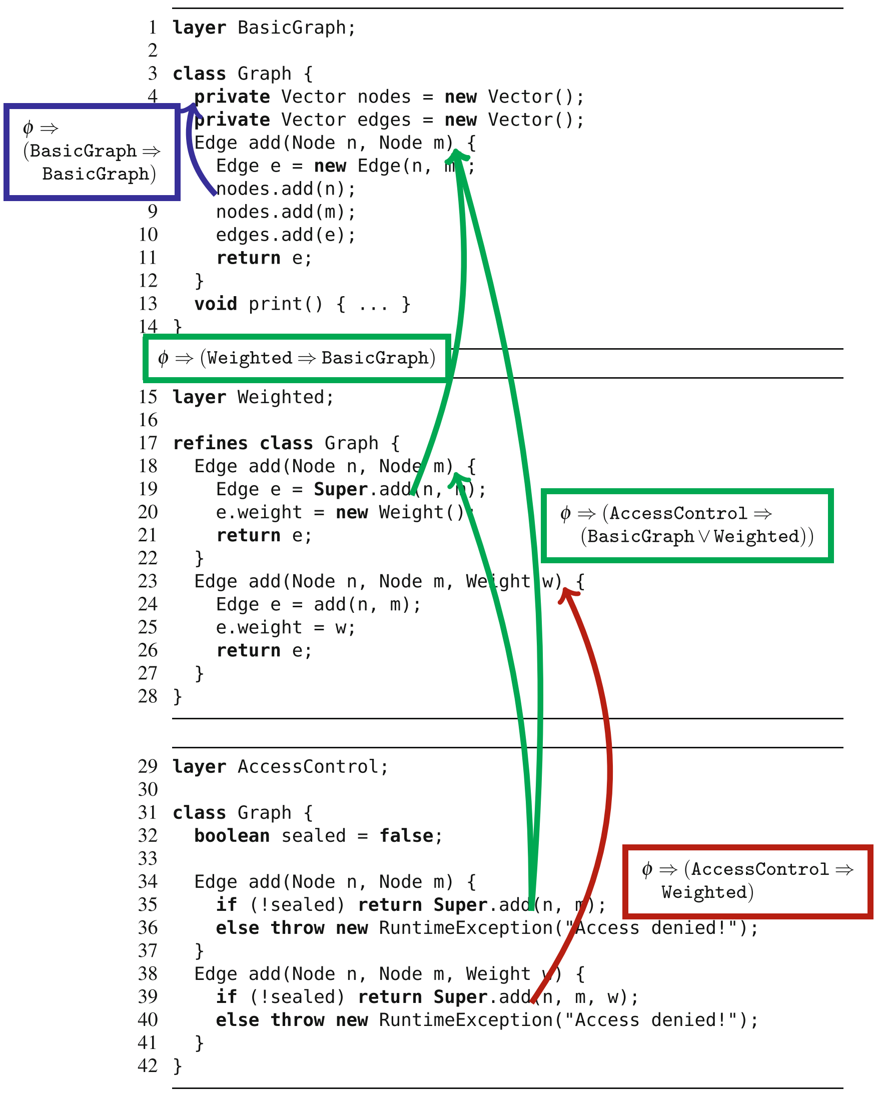
A family-based type checker can be sound and complete with regard to the brute-force approach, but also unsound or incomplete approximations are possible, to simplify implementation or improve the performance of the analysis (still useful to find some errors early in domain artifacts and enforce consistent use of variability implementations).

10.3.4 Family-Based Type Checking for Feature-Oriented Programming

To illustrate the generality of lifting analyses, let us investigate family-based type checking also for feature-oriented programming. The basic mechanism is similar to that for preprocessor implementations: we look up all possible targets of method invocations, field accesses, class references, and so forth. Subsequently, we check reachability constraints and redeclaration errors with presence conditions as before. There are two main differences, though. First, presence conditions for code structures are easily identifiable: All code structures with a feature module have the same presence condition (see Sect. 10.2.3, p. 257). Second, we have a new (extended) language and need to perform different kinds of lookups, some of which are local to the feature module and some of which cross feature module boundaries.

Let us extend the graph example once more as shown in Fig. 10.13: In addition to the basic graph and the extension for feature Weighted from Fig. 6.4 (p. 134),

[6] We do know of cases of “product line” development where this is not so. The situation arises when different versions of a common system are produced in version control by branching code bases that are never merged. We strongly recommend against this practice.


$\phi \Rightarrow$ (BasicGraph $\Rightarrow$ BasicGraph)
$\phi \Rightarrow$ (Weighted $\Rightarrow$ BasicGraph)
$\phi \Rightarrow$ (AccessControl $\Rightarrow$ (BasicGraph $\vee$ Weighted))
$\phi \Rightarrow$ (AccessControl $\Rightarrow$ Weighted)

```
 1 layer BasicGraph;
 2
 3 class Graph {
 4   private Vector nodes = new Vector();
     private Vector edges = new Vector();
     Edge add(Node n, Node m) {
       Edge e = new Edge(n, m);
       nodes.add(n);
 9     nodes.add(m);
10     edges.add(e);
11     return e;
12   }
13   void print() { ... }
14 }
```

```
15 layer Weighted;
16
17 refines class Graph {
18   Edge add(Node n, Node m) {
19     Edge e = Super.add(n, m);
20     e.weight = new Weight();
21     return e;
22   }
23   Edge add(Node n, Node m, Weight w) {
24     Edge e = add(n, m);
25     e.weight = w;
26     return e;
27   }
28 }
```

```
29 layer AccessControl;
30
31 class Graph {
32   boolean sealed = false;
33
34   Edge add(Node n, Node m) {
35     if (!sealed) return Super.add(n, m);
36     else throw new RuntimeException("Access denied!");
37   }
38   Edge add(Node n, Node m, Weight w) {
39     if (!sealed) return Super.add(n, m, w);
40     else throw new RuntimeException("Access denied!");
41   }
42 }
```

Fig. 10.13 Checking whether references to add are well-typed in *all* products

we add a new optional feature AccessControl. Feature AccessControl can prevent users from adding additional edges. We type check this program with a similar strategy as before:

- In Line 8, we access field nodes. The field is defined locally in Line 4 in the same feature module. Thus, the presence conditions of source and target are the same, and the reachability constraint is trivially a tautology:

$$\phi \Rightarrow \big(\mathsf{BasicGraph} \Rightarrow \mathsf{BasicGraph}\big)$$

- In feature module Weighted, we refine method add(Node, Node) of class Graph. Since we use a Super call, we require that a prior declaration of the method exists. A lookup across module boundaries finds a possible target in feature module BasicGraph.[7] Hence, we derive the following reachability constraint:

$$\phi \Rightarrow \big(\mathsf{AccessControl} \Rightarrow \mathsf{BasicGraph}\big)$$

- In feature module AccessControl, we refine method add(Node, Node) once more. This time, we find two possible targets, in feature modules BasicGraph and Weigthed, leading to the following constraint:

$$\phi \Rightarrow \big(\mathsf{AccessControl} \Rightarrow (\mathsf{BasicGraph} \vee \mathsf{Weighted})\big)$$

- Similarly, we refine method add(Node, Node, Weight), but only with one possible target in feature module Weighted. Thus, we add the constraint:

$$\phi \Rightarrow \big(\mathsf{AccessControl} \Rightarrow \mathsf{Weighted}\big)$$

  This constraint can be stricter than a developer might have assumed. In fact this is an instance of the optional-feature problem discussed in Sect. 9.3.

The interesting point is that we can check some constraints *within* a single feature module. Although we cannot compile feature modules in isolation like plug-ins, we still exploit the locality of feature modules. Recently, researchers have started to exploit this locality further in several approaches and even declare or infer corresponding feature interfaces to enable plug-in-like modular type checking despite crosscutting implementations (Apel and Hutchins 2010; Delaware et al. 2009; Schaefer et al. 2011; Kästner et al. 2012b).

10.3.5 Family-Based Analysis with Variability Encoding

When discussing refactorings in Chap. 8, we already mentioned the possibility of variability-preserving rewrites between different variability implementations to change binding times (see Sect. 8.2.3, p. 201). For example, within some limits, we can rewrite a preprocessor-based implementation into one using parameters or feature-oriented programming and vice versa. Where available, we can exploit such rewrites for variability-aware analysis. For example, instead of developing a new variability-aware type system for feature-oriented programming, we could provide

[7] Here, we assume a fixed order of feature modules. A lookup is performed only in previous feature modules. If we want to type check feature modules with a flexible composition order, we need a more sophisticated encoding that reasons about the composition order as well.

```
class Node {
  int id = 0;
  private String name;
  String getName() {
      if (Conf.NAME) return name;
      if (Conf.NONAME) return String.valueOf(id);
      throw VariabilityException();
  }
  Color color = new Color();

  void print() {
    if (Conf.COLOR && Conf.NAME)
        Color.setDisplayColor(color);
    System.out.print(getName());
  }
}

class Color {
  static void setDisplayColor(Color c){/*...*/}
}
```

Fig. 10.14 Possible variability encoding of the graph example from Fig. 10.10; conditionally executed code is highlighted

an automated rewrite that transforms feature-oriented programs into preprocessor-based implementations and type checks them.

Especially for analyses in the course of model checking, rewrites from compile-time variability into run-time variability using parameters are common. The process is called *configuration lifting* or *variability encoding* (Post and Sinz 2008; Apel et al. 2013b).

In Fig. 10.14, we show an example of a possible variability encoding for the graph example of Fig. 10.10. The presence and absence of the features Name, NoName, and Colored is modeled by three corresponding Boolean variables, located in class Conf. Code that is specific to particular features is executed conditionally based on the values of these variables (highlighted in Fig. 10.10). Using standard testing, symbolic execution, model checking, or other existing analysis techniques, we can find that a variability exception is raised when neither Name nor NoName is selected, which indicates an error in the product line (provided this selection is valid according to its feature model).

10.3.6 Feature-Based Analysis Strategies

A Grand Challenge of variability-aware analysis is to analyze features in isolation. Black-box frameworks are especially interesting because their plug-ins can be compiled separately (see Sect. 4.3, p. 79). Separate compilation implies that each plug-in can be type checked in isolation, by compiling against the plug-in interface of the framework. Thus, type errors are detected locally within a plug-in without considering other plug-ins.

However, separate compilation does not yet ensure that all combinations of these plug-ins can be *loaded*. We still need to ensure that plug-ins and the framework share the same interface. Furthermore, there may dependencies between plug-in interfaces, and there could be constraints on which and how many plug-ins may be loaded. For example, we might want to guarantee that in every product at least one (or at most one, or exactly one) plug-in is installed. That is, some checks are still required at composition time.

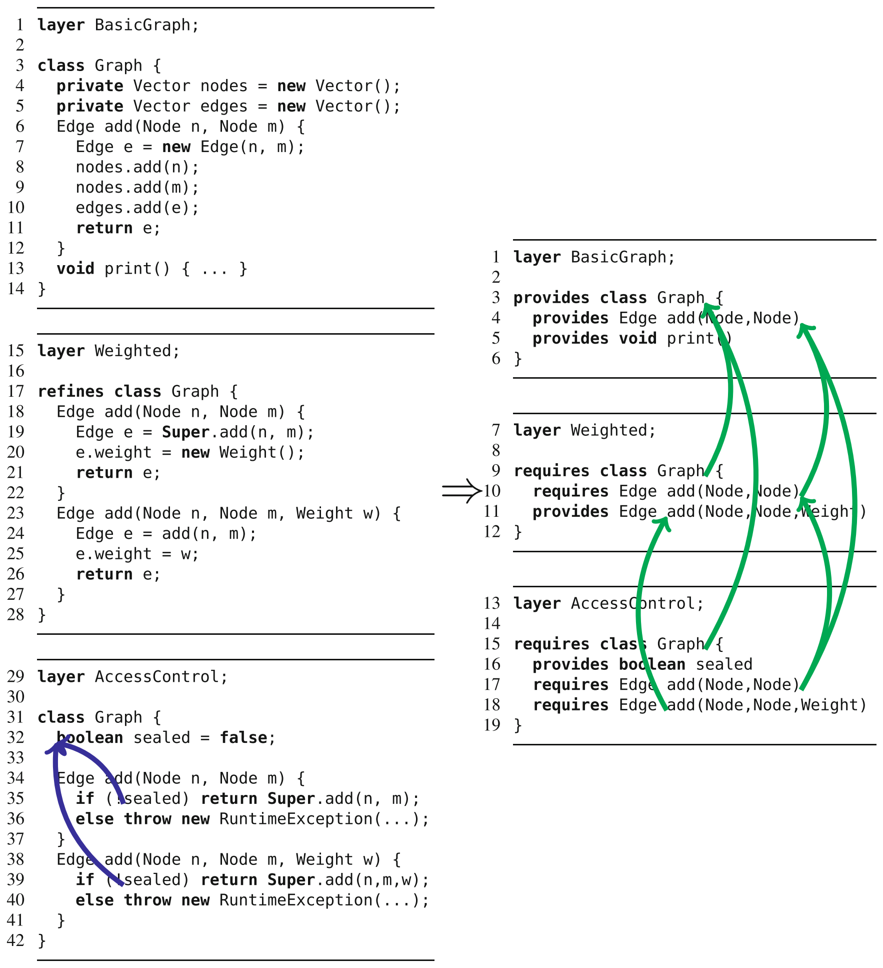
Also, in feature-oriented programming (and aspect-oriented programming and delta-oriented programming) modular type checking has been explored. The idea is to type check a feature module in isolation as far as possible. As we have seen previously in Fig. 10.13, many checks can be performed locally within a feature module. Checks that are not performed locally, can be deferred to composition time. That is, constraints referring to code fragments of other features can be expressed (explicitly or inferred) in an interface. An interface constraint of a feature module might specify that it requires some other feature module to provide a class, method, or field. The interface also describes which structures are exported, so they can be used by other features. Compatibility between modules is then checked at composition time (usually called linker checks). In Fig. 10.15, we exemplify this idea by means of our previous graph example.

An exponential number of possible feature selections and corresponding module compositions remains, for which we need to check interface compatibility. Each compatibility check is cheaper than rechecking the entire source code of the product though. To aim for complete coverage of all feature selections, while avoiding a brute-force approach, we can again use sampling or a family-based approach that checks reachability constraints between interfaces, as illustrated in Fig. 10.15.

Feature-based analysis enables an *open-world* development strategy where not all features may be known at development time or analysis time. For example, when extending a framework, plug-in developers may not know about all other plug-ins in the system. It can be a good strategy to first check plug-ins in isolation as far as possible and then check plug-in compatibility when actually composing specific plug-ins. Open-world development becomes increasingly important with software ecosystems to which multiple independent parties contribute (Bosch 2009). In contrast, the family-based strategies discussed previously require that all features are known at analysis time, it requires a *closed-world* scenario.

10.3.7 Beyond Type Checking

So far, we have illustrated different analysis strategies by means of type checking. The outlined strategies can be applied to other kinds of analyses as well. In all cases, the idea is to lift an existing analysis to check a given property for the entire product line. If possible, we want to move beyond brute-force and sampling approaches. So far, researchers have investigated variability-aware parsing (Kästner et al. 2011; Gazzillo and Grimm 2012), variability-aware data-flow, control-flow, and information-flow



```
layer BasicGraph;

class Graph {
  private Vector nodes = new Vector();
  private Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  void print() { ... }
}
```

```
layer Weighted;

refines class Graph {
  Edge add(Node n, Node m) {
    Edge e = Super.add(n, m);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = add(n, m);
    e.weight = w;
    return e;
  }
}
```

```
layer AccessControl;

class Graph {
  boolean sealed = false;

  Edge add(Node n, Node m) {
    if (!sealed) return Super.add(n, m);
    else throw new RuntimeException(...);
  }
  Edge add(Node n, Node m, Weight w) {
    if (!sealed) return Super.add(n,m,w);
    else throw new RuntimeException(...);
  }
}
```

```
layer BasicGraph;

provides class Graph {
  provides Edge add(Node,Node)
  provides void print()
}
```

```
layer Weighted;

requires class Graph {
  requires Edge add(Node,Node)
  provides Edge add(Node,Node,Weight)
}
```

```
layer AccessControl;

requires class Graph {
  provides boolean sealed
  requires Edge add(Node,Node)
  requires Edge add(Node,Node,Weight)
}
```

Fig. 10.15 References to field sealed can be checked entirely within feature AccessControl (*left*); references to the `add` methods and the class Graph cut across feature boundaries and are checked at composition time based on the features' interfaces (*right*)

analysis (Brabrand et al. 2012; Bodden 2012; Liebig et al. 2012), variability-aware testing, mostly based on sampling (Cohen et al. 2007; Oster et al. 2010; Kästner et al. 2012c; Kim et al. 2012), variability-aware model checking (Li et al. 2005; Post and Sinz 2008; Classen et al. 2010, 2012; Apel et al. 2013b), variability-aware theorem proving (Thüm et al. 2011b; Thüm et al. 2012b), and variability-aware consistency checking of models (Czarnecki and Pietroszek 2006).

We will not go into details of these approaches, but there seem to be repeating patterns. A general strategy is to perform analysis on shared code only once and to reason about entire configuration spaces by means of propositional formulas and

SAT solvers. For the interested reader, we recommend a survey of variability-aware analysis (analysis strategies, specification strategies, and classification of existing analyses) by Thüm et al. (2012a).

10.4 Case Studies and Experience

Analysis of product lines is a comparably new research area, and most results are from academic contexts. Nevertheless, we want to highlight some achievements and share some results to give an impression of what product-line analysis is capable of.

Regarding analysis of feature models, early product configurators were hard to use and allowed people to configure invalid products or get stuck in the configuration process. Modern configurators, also of commercial product-line tools, are quite advanced, thanks to feature-model analysis. Partial selections are rapidly propagated and conflicts are explained (see tooling section below). Researchers have found that the tools scale interactive configuration easily to feature models with several hundreds or even thousands of features.

Tartler et al. (2011) have analyzed the feature-to-code mapping of the Linux kernel in detail with the goal of finding inconsistencies, especially dead code. To this end, they reverse engineered the feature-modeling language Kconfig (see Sect. 2.3.6, p. 36) and extracted presence conditions from Linux's build system *Kbuild* (see Sect. 5.2.3, p. 107) and Linux's preprocessor-based implementation. They found 117 incorrect mappings between features and code fragments, where `#ifdef` constructs referred to features that are not declared in the feature model (typically, typos such as `CONFIG_CPU_HOTPLUG` instead of `CONFIG_HOTPLUG_CPU`). Following the approach outlined in Sect. 10.2.1, they found over 1,000 dead code fragments and manually proposed 214 patches to the Linux community, of which a majority was accepted to be included into the kernel. They classify 22 of those dead code fragments as actual bugs that change the behavior of the kernel in unexpected ways. The analysis is fast and can analyze the entire kernel in about 15 min. Challenges arise mostly from the difficult extraction of information from the feature model and the build system (due to subtle semantic details and anachronisms of the language), so the analysis is not entirely precise. Overall, this project impressively shows how even lightweight analyses can discover many problems, even in well-developed and peer-reviewed code of the Linux kernel.

Also variability-aware analysis, especially type checking, was applied to a series of larger projects and discovered many implementation bugs. Notable studied systems are AHEAD itself (70 features; 48k lines of composition-based Jak code; Thaker et al. 2007), Mobile RSS Reader (14 features, 20k lines of annotation-based Java code; Kästner et al. 2012a), Mobile Media (14 features; 6k lines of annotation-based Java code; Kästner et al. 2012a), Busybox (811 features; 260k lines of annotation-based C code; Kästner et al. 2012b), and the x86 Linux kernel (7000 features; 6.7M lines of annotation-based C code). In all projects, bugs were found: conflicting introductions of a method in multiple modules in AHEAD, dangling calls across

```
 1 //... skipped 260 lines
 2 struct globals {
 3         double   cur_time;
 4         //... skipped 11 lines
 5 #if ENABLE_FEATURE_NTPD_SERVER
 6         int      listen_fd;
 7 #endif
 8         unsigned verbose;
 9         //... skipped 73 lines
10 };
11
12 //... skipped 1761 lines
13
14 int ntpd_main(int argc UNUSED_PARAM, char **argv)
15 {
16 #undef G
17         struct globals G;
18         //... skipped 81 lines
19         if (i > (ENABLE_FEATURE_NTPD_SERVER && G.listen_fd != -1)) {
20             ...
21         }
22         ...
23 }
```

Fig. 10.16 Variability-related bug in Busybox: When feature NTPD_SERVER is deactivated, field listen_fd is removed from struct globals. but still accessed in Line 19 (ENABLE_FEATURE_NTPD_SERVER is a macro defined to either `0` or `1` depending on the feature selection)

feature boundaries in Mobile RSS Reader, a missing dependency in the feature model and incorrectly annotated import statements in Mobile Media, and dangling references in Busybox. In Fig. 10.16, we exemplify a bug found in Busybox, which is hard to find manually. In all cases, performance is in the realm of analyzing less than ten sampled products.

Experience with variability-aware static analysis (Brabrand et al. 2012; Bodden 2012; Liebig et al. 2012) and variability-aware model checking (Li et al. 2005; Post and Sinz 2008; Classen et al. 2012; Apel et al. 2013b) is similar, but tools in this field are just starting to approach larger scale studies.

Overall, experience shows that efficient analysis of entire product lines is possible and useful. Analysis finds real bugs and can be performed in reasonable time. Difficulties typically stem from undisciplined implementation strategies and legacy artifacts (for example, extracting presence conditions from build systems and lexical preprocessors and reverse engineering feature modeling languages), whereas the analysis is typically straightforward.

10.5 Tooling

Analysis of feature models has matured and some analyses are now available even in commercial product-line tools, such as *pure::variants*. The *SPLOT* website[8] offers the possibility to try many analyses directly online. *FeatureIDE* integrates many

[8] http://www.splot-research.org/

feature-model analyses. The *FAMA*[9] tool suite is probably the most comprehensive selection of different analyses available right now. *FAMA* also allows selecting from a large range of different solvers.

For checking the feature-code mapping only few tools are readily available. Specifically for the Linux kernel, the *Undertaker*[10] tool analyzes the mapping with the goal to dead (and undead) code fragments. Some implementation approaches ensure directly that only features declared in the feature model are referenced in the implementation; examples are *CIDE*[11] and *FeatureMapper*.[12]

For variability-aware analysis, almost only concept and research prototypes are available. Some tools that can be used for experimentation are *SafeGen*,[13] *TypeChef*,[14] *CIDE*,[15] *CIDE+*,[16] *SPLverifier*,[17] *VMC*,[18] and *ProVeLines* and *SNIP*.[19]

10.6 Further Reading

Analyses of feature models are well explored in the literature. Batory (2005), Benavides et al. (2005), and van der Storm (2004) were among the first to describe encodings of feature models as propositional formulas to reason about them with SAT solvers, solvers for constraint satisfaction problems, and binary decision diagrams. Benavides et al. (2010) provide an excellent overview of developments in the field, including many analysis questions and different implementation strategies and tools. They also provide a good introduction of how to reason about feature models in the presence of non-Boolean features and constraints.

A good example of analysis of the feature-code mapping is the *Undertaker* project by Tartler et al. (2011). The authors describe in detail the challenges of extracting feature models and presence conditions and their experience with reporting bugs to the developer community. An earlier and simpler approach was described by Metzger et al. (2007) who provided a separate variability model for each implementation artifact (instead of extracting presence conditions from some implementation), and subsequently checked intended variability against the variability modeled for the implementation.

[9] http://www.isa.us.es/fama/

[10] http://vamos.informatik.uni-erlangen.de/trac/undertaker

[11] http://fosd.net/cide

[12] http://featuremapper.org/

[13] http://www.cs.utexas.edu/~schwartz/ATS.html

[14] http://ckaestne.github.com/TypeChef/

[15] http://fosd.net/CIDE

[16] http://homepages.dcc.ufmg.br/~mtov/cideplus/

[17] http://fosd.net/FAV

[18] http://fmtlab.isti.cnr.it/vmc/

[19] http://www.info.fundp.ac.be/fts/

Sampling strategies have been first explored outside the product-line context as combinatorial testing, but have quickly been applied to product lines as well. There is a large body of research to which we can only provide initial pointers (Cohen et al. 2007; Oster et al. 2010; Perrouin et al. 2010).

The idea to analyze the domain artifacts of the entire product line originated initially from work on generators (Huang et al. 2005) and checking model consistency (Czarnecki and Pietroszek 2006). The first type checking approach for product line was proposed by Thaker et al. (2007). The field of variability-aware analysis has recently exploded with research contributions from different fields. Readers interested in this field may follow the references in Sect. 10.3.7 as a starting point. Also, a recent survey by Thüm et al. (2012a) provides a good overview of the field and the different strategies applied.

Work on feature-based analysis often has striking parallels with research in programming languages, regarding modularity and module systems. The goal is the same: Check errors locally and early to allow development in an open-world style. Again, we can only provide initial pointers to a large body of research (Leroy 1994; Cardelli 1997; Blume and Appel 1999; Ancona and Zucca 2001; Ancona et al. 2005; Strniša et al. 2007).

Finally, there is plenty of work on product-line testing. Unfortunately, testing cannot yet exploit the similarities between products as static variability-aware analyses do, but relies more on sampling. Typical technical strategies are to test domain artifacts in isolation as far as possible and to prepare reusable test cases as part of domain engineering that can be executing during application engineering. Pohl et al. (2005) provide a good overview of basic testing strategies and Engström and Runeson (2011) and da Mota Silveira Neto et al. (2011) have conducted recent surveys of product-line testing that provide good starting points for further reading.

Exercises

10.1. When are analyses of software product lines useful or even necessary? Discuss opportunities and challenges. Which phases of the product-line-development process can be supported by analyses? Provide illustrative examples to explain your position.
10.2. Analyze (i) the feature models in Fig. 10.17, (ii) the feature models of the graph example in Fig. 2.6, and (iii) the feature models created in Exercises 2.4 and 2.5 (p. 43) as follows:

(a) Translate the feature model into a propositional formula.
(b) Provide two valid and two invalid feature selections (if possible).
(c) Check whether the feature model is consistent.
(d) Provide two assumptions that hold in the feature model and two assumptions that do not hold. Select assumptions that could be reasonably used as tests.
(e) Detect whether the feature model contains any dead or false optional features.

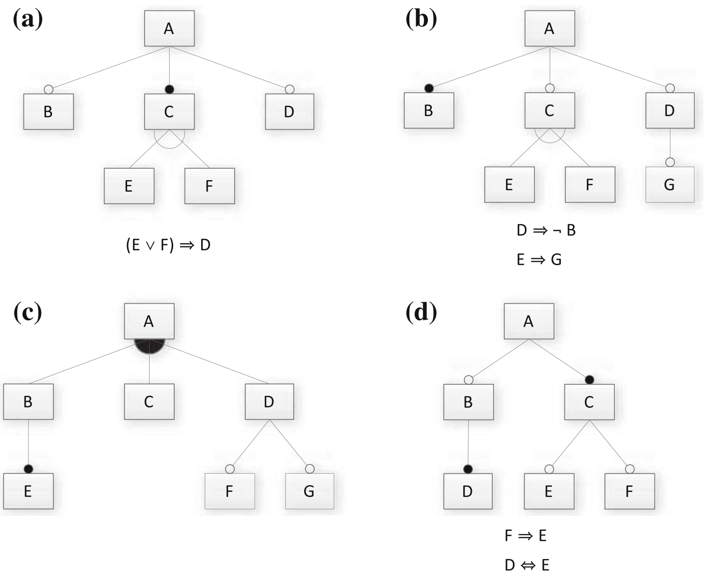


Fig. 10.17 Sample feature models

(f) Illustrate constraint propagation on a partial feature selection (if possible). As partial feature selection use the last two features of the valid feature selections of Exercise 10.2b)
(g) Calculate the number of valid feature selections (you may ignore cross-tree constraints).
(h) Perform a change of the feature model that is (i) a refactoring, (ii) a generalization, (iii) a specialization, and (iv) none of the above. Demonstrate that the change actually falls into the given category.

10.3. Build an infrastructure to answer the questions of Exercise 10.2 mechanically. Define a simple textual (or XML) format for feature models; translate feature models into propositional formulas; answer the questions by translating them into Boolean satisfiability problems; solve the satisfiability problems by handing them over an off-the-shelf SAT solver as sat4j or MiniSat[20]; and print the solution.

10.4. In the context of the domains from Exercise 2.5 (p. 43), discuss when optimizing feature selections may be useful or necessary. Which nonfunctional requirements may be worth to optimize? Which functional requirements may be implemented by different features with different nonfunctional trade-offs? Provide illustrative examples.

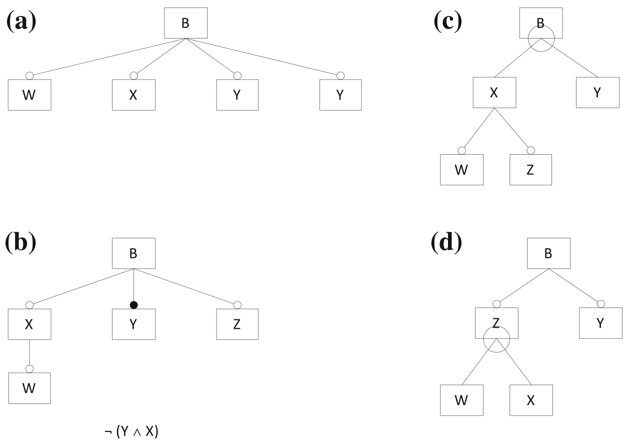
10.5. Discuss how you could extend your analysis infrastructure from Exercise 10.3 to support constraints over non-Boolean feature attributes and optimization goals. For example, assume each feature has a known price and a known impact on binary size

[20] http://www.sat4j.org/, http://minisat.se/

and you want to complete a partial feature selection such that the resulting product is smaller than 500 kb and has the lowest possible price. Investigate what technology could be used to perform such optimization problem.

10.6. Derive presence conditions for each code fragment in the following file. Subsequently determine which code fragments are dead code fragments given the four feature models:

```
 1 a = 1; b = 1;
 2 #ifdef X
 3 a++;
 4 #ifdef Y
 5 a = a * 2;
 6 #endif
 7 #endif
 8 #ifdef Z
 9 b = 4;
10 #ifdef X
11 a = a - 2;
12 #elif W
13 b = b - 1;
14 #else
15 b = 5;
16 #endif
17 b = b / a;
18 #endif
```



10.7. To test a product line, we want to pursue a sampling strategy.

(a) Discuss possible coverage criteria. Which coverage would be necessary to detect the division-by-zero in Line 17 of the example in Exercise 10.6?
(b) Collect a small set of feature selections to fulfill the following coverage goals:

  (i) Feature coverage: Each feature of the product lines described by the feature models in Fig. 2.6 and in Fig. 10.17a should be included in at least one feature selection.
  (ii) Feature-code coverage: Each line of code in the code example of Exercise 10.6 should be included in at least one feature selection (not considering any feature models).
  (iii) Feature-code coverage: Each line of code in the code example of Exercise 10.6 should be included in at least one feature selection that is also valid in the corresponding feature models.
  (iv) Pair-wise coverage: In a product line with five optional and independent features A, B, C, D, and E, for every pair of features (f,g), there should be a feature selection with f and g, one with f and without g, and one without f but with g.
  (v) Pair-wise coverage: For the feature model of the graph example in Fig. 2.6 and for the feature model in Fig. 10.17a–c, achieve pair-wise coverage as in the previous task.

10.8. Explain the strategy of family-based type checking on the following two code examples. What reachability constraints can be derived from the code base? For which feature selections will the code not compile? Provide a feature model that describes all compilable products.

(a) A simple hello-world program with three features World, Bye and Slow, declared as optional and independent.

```
import <<Java standard library>>;

class HelloWorld {

  //#ifdef WORLD
  static String msg = "Hello World\n";
  //#endif

  //#ifdef BYE
  static String msg = "Bye bye!\n";
  //#endif

  public static void main(String [] args) throws Exception {
    //#if defined(SLOW) && defined(WORLD)
    Thread.sleep(1000);
    //#endif
    System.out.println(msg);
  }
}
```

(b) A simple object-oriented store with two alternative base features (SingleStore and MultiStore) and an optional feature AccessControl implemented with feature-oriented programming.

```
layer SingleStore;

class Store {
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

```
layer MultiStore;

class Store {
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

```
layer AccessControl;

refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super().read(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return Super().readAll(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super(Object).set(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

10.9. Provide examples of analyses for product lines that are not sound or complete with regard to what an analysis using a brute-force approach would find. Discuss in which scenarios such analyses may still be useful.

10.10. *Advanced task, requires a background in type systems* (Pierce 2002). Design a formal type system for product lines based on the simply typed lambda calculus.

We start with an extended version of the lambda calculus enhanced with compile-time variability annotations:

$$\mathtt{e} = \mathtt{x} \;\mid\; \lambda\mathtt{x}:\tau.\,\mathtt{e} \;\mid\; \mathtt{e}\;\mathtt{e} \;\mid\; \mathtt{c} \;\mid\; \Upsilon\mathtt{f}.\,\mathtt{e} - \mathtt{e}$$

$\Upsilon\mathtt{f}.\,\mathtt{e} - \mathtt{e}$ represents a compile-time choice between two expressions depending on feature $\mathtt{f}$. By evaluating compile-time choices Υ with a feature selection, we can derive a traditional lambda-calculus expression.

Design a type system for the variability-enhanced lambda calculus, such that an variability-enhanced expression is well-typed if and only if all derivable lambda-calculus expressions are well-typed (see Fig. 10.9). Proof soundness and completeness with regard to the brute-force approach.

Appendix A
Tool Support

Throughout the book, we have seen several examples of tools for product-line development, from simple preprocessors to sophisticated composition tools and compilers. In this chapter, we introduce a number of tools for practical product-line development or useful for teaching. We do not intend to provide a comprehensive overview or discuss the pros and cons of specific commercial or academic tools—the field is too broad and continuously changing. Instead, we give brief recommendations of available tools, where we focus on freely available and stable tools that left the status of early academic prototypes.

A.1 Overview

As introduced in Chap. 2, product-line development consists of multiple phases, each of which shall be supported by proper tools.

- *Domain analysis.* During domain analysis, tools shall support the creation of and reasoning about feature models. They shall offer facilities to manage scoping decisions and document other concerns.
- *Domain implementation.* Depending on the modeling or implementation mechanism, editor support as known from modern IDEs like Eclipse shall support domain implementation. Furthermore, tools shall support different kinds of mappings between features and development artifacts.
- *Requirements analysis.* During requirements analysis, tools shall guide developers when selecting desired features, and verify the correctness or completeness of a selection with regard to the feature model.
- *Product derivation.* Again, depending on the implementation mechanisms, compilers, preprocessors, or composition engines shall automate the derivation process.

For each of these phases, proper tool support is desirable. However, there must also be tools that integrate multiple or all phases of product-line development. Integration allows tool support to cross phases, for example, to propagate the renaming of a feature from the feature model to the implementation mapping and to all existing feature selections. Given the variety of different mechanisms in each phase, the mixture of mechanisms in practice, and the demand to process different artifacts uniformly with regard to variability (see Sect. 3.2.6, p. 62), most product-line tools are extensible and try to connect the phases.

A.2 Commercial Tools

At the time of writing there are two major players on the market for product-line tools *pure-systems GmbH* with *pure::variants* and *BigLever Software, Inc* with *Gears* (and several more regarding processes and consulting, which are not in our focus here). Both tools are used in industrial practice by many companies. Technically, both are extensible frameworks that cover all phases of the process of product-line development.

A.2.1 pure::variants

pure::variants is a commercial tool developed and marketed by the German company *pure-systems GmbH* (`http://www.pure-systems.com/`). It integrates all phases of product-line development and can be used with different implementation mechanisms (they key concepts are independent of any specific language or tool). pure::variants integrates into Eclipse and can be extended with additional plug-ins.

For domain analysis, pure::variants uses feature models very close to the notation introduced in Sect. 2.3. For large-scale models (with hundreds or thousands of features), pure::variants provides scalable tree-based editors and simple facilities to decompose feature models. Features can be enhanced with all kinds of annotations and parameters (only briefly discussed in Sect. 2.3) and logical rules can be used to express even complex non-Boolean constraints. There are connectors to various requirements-engineering tools. An editor for the requirements-engineering phase to select features is tightly integrated and supports various kinds of feature-model analyses (see Sect. 10.1, p. 260).

For the mapping between problem and solution space, pure::variants provides a generic component model, the *component family model*. This model relates individual components to features. The term component is used in a broad sense and refers to a set of configurable functionalities, ranging from classes and aspects to compiler flags. In this context, pure::variants works like a sophisticated build system, in which developers can specify how and when to process artifacts and with which compilers, preprocessors, or composition tools. In addition to preconfigured scenarios for C/C++ and Java programs, pure::variants also ships with a customizable preprocessor for conditional compilation in arbitrary artifacts and can be extended with connectors, for example, for various modeling and testing tools.

Earlier in this book, in Fig. 2.10 (p. 38), we have shown a screenshot of the pure::variant's workbench for our graph example, including editors for feature models and family models. A community edition of pure::variants, limited to comparably small models, is freely available for experimentation.

A.2.2 Gears

Gears is a product-line tool of *BigLever Software, Inc.* (`http://www.biglever.com/`). Also, Gears aims at automating product derivation starting from a feature selection. Gears supports both feature models and simple lists of configuration parameters. Products are configured by selecting features or values for configuration parameters.

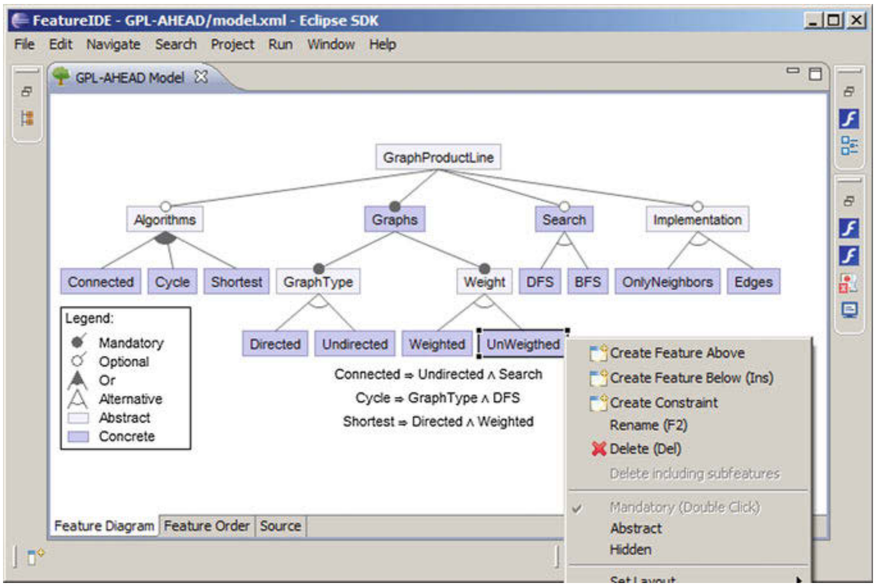
Given a configuration, Gears acts as a sophisticated build system that can run compilers, preprocessors, and composition tools, as specified by the developer. Next to calling external tools, standard examples are to use conditional compilation in various artifacts (a generic preprocessor is provided) or to inline the content of feature-specific files at marked locations in other files (named variation points). This way, building products from reusable artifacts can be automated entirely.

In addition, Gears is highly extensible. Connectors for integrated development environments (such as Eclipse), requirements engineering tools, modeling tools, word processors, and others exist. Gears provides an API so that tool builders can connect to a single central variability-management mechanism.

A.3 FeatureIDE: An Open-Source Tool for Product-Line Implementation

While the commercial tools provide flexible production-quality solutions for industrial product-line development, researchers and educators might look for open-source solutions, easy for experimentation, extension, and class-room usage. FeatureIDE is an open-source development environment for product lines, targeted primarily at researchers, teachers, and students (`http://fosd.net/fide`). FeatureIDE integrates closely with several research tools, such as the AHEAD tool suite, FeatureC++, FeatureHouse, AspectJ, DeltaJ, and Java preprocessors. It is extensible using Eclipse's plug-in mechanism.

For domain analysis, FeatureIDE provides a graphical feature-model editor, as shown in Fig. A.1, which incorporates several analysis techniques (see Sect. 10.1, p. 248). The ability to produce high-quality graphics of feature models (for example, for teaching and research publications) was given precedence over scalability of the graphic editor. Also, the support for extra annotations and non-Boolean parameters is restricted compared to the commercial tools. Feature modeling is tightly integrated with the feature selection of the requirements-



Connected ⇒ Undirected ∧ Search
Cycle ⇒ GraphType ∧ DFS
Shortest ⇒ Directed ∧ Weighted

Fig. A.1 The feature-model editor of FeatureIDE

analysis phase (see Fig. 10.3 on p. 276; supporting reasoning about partial configurations, propagating feature renaming, updating feature selections when feature model constraints change, and so forth). The research nature is also visible by the fact that FeatureIDE provides several import and export mechanisms for feature models specified by several research tools.

For the solution space, FeatureIDE supports several specific tools, currently the AHEAD tool suite (Jak), FeatureC++, FeatureHouse, AspectJ, DeltaJ, and Java preprocessors. FeatureIDE is extensible by writing plug-ins, but, in contrast to the commercial tools, it does not provide general-purpose build-system mechanisms that would allow it to call arbitrary other tools. The tighter integration of specific implementation strategies (with plug-ins) provides better editor support and simplifies the learning curve (as most complexity from the build process is hidden).

For AHEAD, FeatureIDE provides an editor for Jak files as illustrated in Fig. A.3 (see also Sect. 6.1.3, p. 139). A tight integration allows several editor services for product lines implemented in Jak: syntax highlighting, automatic generation and compilation in the background, error reporting (traced back from Java errors on the composed files), and sophisticated visualizations such as the collaboration diagram shown in Fig. A.2 (see also Sect. 6.1.1, p. 136). We again see integration with other phases: Features are mapped to feature modules by name, and feature renaming propagates to the solution space, products are automatically recompiled when changing the feature selection, and so forth. The other mentioned languages are similarly deeply integrated; when languages come with

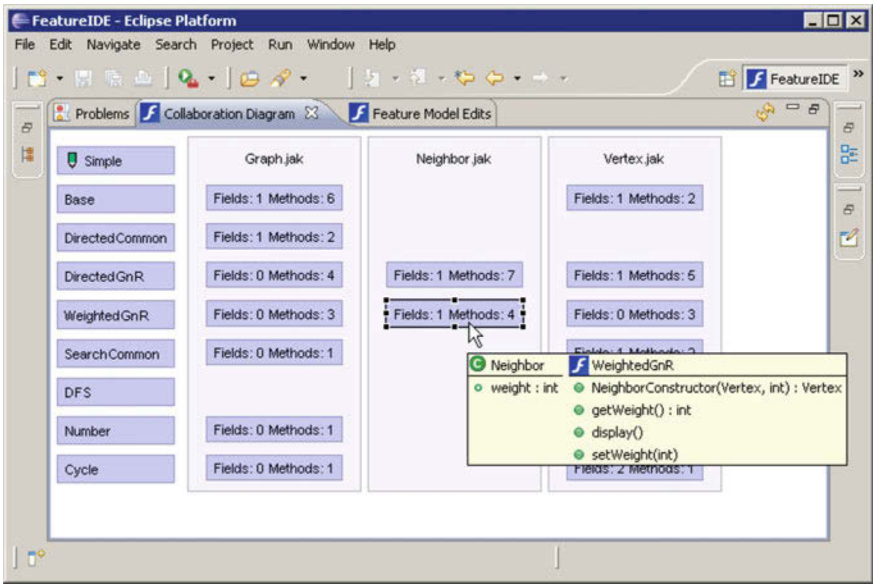
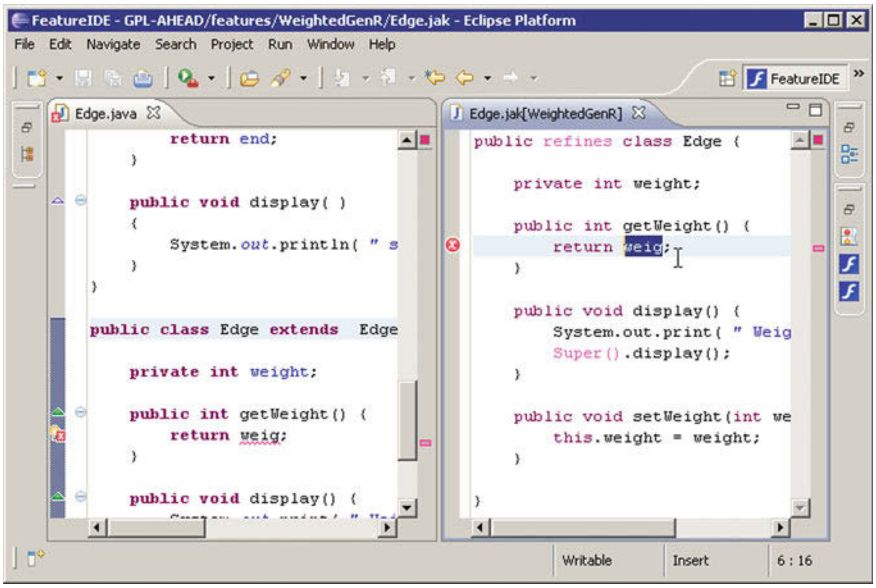


Fig. A.2 FeatureIDE: the collaboration diagram view


FeatureIDE - GPL-AHEAD/features/WeightedGenR/Edge.jak - Eclipse Platform
File Edit Navigate Search Project Run Window Help
FeatureIDE
Edge.java
Edge.jak[WeightedGenR]
Writable
Insert
6 : 16

```
        return end;
    }

    public void display( )
    {
        System.out.println( " s
    }
}

public class Edge extends   Edge

    private int weight;

    public int getWeight() {
        return weig;
    }

    public void display() {
```

```
public refines class Edge {

    private int weight;

    public int getWeight() {
        return weig;
    }

    public void display() {
        System.out.print( " Weig
        Super().display();
    }

    public void setWeight(int we
        this.weight = weight;
    }

}
```

Fig. A.3 FeatureIDE: A syntax error in a class refinement in Jak

their own editor, such as C++ and AspectJ, these are reused.

FeatureIDE is not necessarily aimed at supporting industrial-scale product-line development. However, it provides a good base for teaching product-line engineering, with feature models (Chap. 2), preprocessors (Chap. 3), and advanced language-based mechanisms (Chap. 6). With its open-source nature based on Eclipse plug-ins it is highly extensible.

An overview on design considerations for FeatureIDE and learned lessons can be found in Thüm et al. (2013).

A.4 Further Tools

In the corresponding parts of the book, we have mentioned and discussed several other tools for product-line development (mostly focused on one specific task). Here, we provide only a brief list of the stable research tools as well as publicly available open-source tools that can be used right away for implementing product lines. More experimental tools can be found in the corresponding chapters of the book. For each tool, we summarize for which task they were designed, where we discussed them in this book, and where to get them.

AHEAD Tool Suite. Collection of composition and supporting tools for feature-oriented programming in Jak and other languages. Includes a tool *SafeGen* for variability-aware type checking. Command-line research tools, partially integrated in *FeatureIDE*. See also Sect. 6.1.
`http://www.cs.utexas.edu/~schwartz/ATS.html`

Antenna. Lexical preprocessor designed for Java ME applications, with integrations in several IDEs, such as NetBeans, Eclipse, and FeatureIDE. See also Sect. 5.3.3.
`http://antenna.sourceforge.net`

AspectJ. Aspect-oriented programming language based on Java with corresponding compiler. A commercial-quality Eclipse-based IDE *AJDT* is available. See also Sect. 6.2.
`http://www.eclipse.org/aspectj/`

CDL. Textual variability modeling language and corresponding tool infrastructure. Originally designed for the *eCos* operating system. See also the description of Berger et al. (2010).
`http://ecos.sourceware.org/docs-2.0/cdl-guide/cdl-guide.html`
`https://code.google.com/p/variability/wiki/CDLTools`

CIDE. Eclipse-based research prototype for virtual separation of concerns. Has been partially reimplemented in several other projects. See also Chap. 7.
`http://fosd.net/CIDE`
`http://fosd.net/fc`
`http://www.dcc.ufmg.br/~mtov/cideplus/`

Clafer. A lightweight yet expressive language for structural modeling: feature modeling and configuration, class and object modeling, and metamodeling. Clafer Tools is Integrated set of tools based on Clafer, supporting model analysis, configuration, and multi-objective optimization, exploration, and visualization.
http://clafer.org

ConcernMapper. Feature mapping and tracing tool, targeted at arbitrary concerns in general software development. See also Sect. 7.1.
`http://www.cs.mcgill.ca/~martin/cm/`

ContextJ. Context-oriented extension of Java. Similar extensions with varying maturity exist for many other languages (*ContextL*, *ContextJS*, *ContextR*, and so forth). See also Sect. 6.6.3.
`http://www.swa.hpi.uni-potsdam.de/cop/`

cpp. Lexical preprocessor part of the C language standard (ISO). Shipped with every C and C++ compiler. See also Sect. 5.3.1.
`http://gcc.gnu.org/`
`http://clang.llvm.org/`, and others

CVL. Common variability language. Upcoming industry standard for variability modeling. Basic tool support is available in the form of Eclipse plug-ins.
`http://www.omgwiki.org/variability/doku.php/doku.php?id=cvl_tool_from_sintef`

DeltaJ. Delta-oriented programming language based on Java and corresponding composition engines. Command-line research tool and Eclipse-based IDE. See also Sect. 6.6.1.
`http://deltaj.sourceforge.net/`

FAMA. Comprehensive research framework for feature-model analysis. See also Chap. 10.
`http://www.isa.us.es/fama/`

FeatureHouse. Composition engine for feature-oriented programming in various languages. Declarative extension mechanisms to plug-in new languages. Command-line research tool, integrated also in *FeatureIDE*. See also Sect. 6.1.
`http://fosd.net/fh`

FeatureMapper. Eclipse editor for product lines of ecore models; annotation-based; supports views and some analysis. See also Sects. 5.3.3 and 7.5.
`http://featuremapper.org/`

git. State-of-the-art distributed version-control system, with advanced branching and merging capabilities. See also Sect. 5.1.
http://git-scm.com/

Kbuild. A collection of build files and conventions that form the build system of the Linux kernel. Part of the Linux kernel, but also used by several other projects. Integrates with *Kconfig*. See also Sect. 5.2.3.
http://www.kernel.org
http://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

Kconfig. Textual variability-modeling language and corresponding configurator. Developed for and distributed with the Linux kernel, but also used by several other projects. See also Sect. 2.3.6 and the description of Berger et al. (2010).
http://www.kernel.org
http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
http://gsd.uwaterloo.ca/feature-models-in-the-wild

Koala. Component infrastructure and composition mechanisms, originally developed by Philips Research for consumer electronics. An open-source implementation is available as well. See also Sect. 4.4.
http://www.program-transformation.org/Tools/KoalaCompiler

Munge. Simple, open-source, lexical preprocessor for Java that does not break existing Java tooling (conditional-inclusion directives are inside comments). See also Sect. 5.3.3.
http://sonatype.github.com/munge-maven-plugin/

OSGi framework. Module system for Java applications that can be used to develop framework and component-based solutions. Underlying technology of the Eclipse project. See also Sects. 4.3 and 4.4.
http://www.osgi.org;
http://www.eclipse.org

SNIP. Variability-aware model-checking tool for product-line models written in the specification language Promela. See also Chap. 10.
http://www.info.fundp.ac.be/fts/

SPLOT. Research tools for editing, collecting, and analyzing feature models. Entirely available as web-based online tools. See also Chap. 10.
http://www.splot-research.org/

SPLverifier. Tool suite for variability-aware model checking of *FeatureHouse*-based product lines written in C and Java. See also Chap. 10.
http://fosd.net/FAV

TypeChef. Research framework of variability-aware analysis of preprocessor-based product lines written in C, such as the Linux kernel; includes variability-aware type checking and data-flow analysis. See also Chap. 10.

`https://github.com/ckaestne/TypeChef`

Undertaker. Tool that analyzes the mapping of preprocessor directives to configuration models, including dead-code detection and other analyses. See also Chap. 10.

`http://vamos.informatik.uni-erlangen.de/trac/undertaker/`

VMC. Variability-aware model-checking tool for product-line models represented as transition systems. See also Chap. 10.

`http://fmtlab.isti.cnr.it/vmc/`

References

Adams B, De Meuter W, Tromp H, Hassan AE (2009) Can we refactor conditional compilation into aspects? In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 243–254

Adams B, De Schutter K, Tromp H, De Meuter W (2007) Design recovery and maintenance of build systems. In: Proc. Int'l Conf. Software Maintenance (ICSM). IEEE Computer Society, pp 114–123

Adams B, De Schutter K, Tromp H, De Meuter W (2008a) The evolution of the Linux build system. Electronic Communications of the EASST, 8

Adams B, Van Rompaey B, Gibbs C, Coady Y (2008b) Aspect mining in the presence of the C preprocessor. In: Proc. AOSD Workshop on Linking Aspect Technology and Evolution (LATE). ACM Press, pp 1–6

Adler C (2010) Optional composition: A solution to the optional feature problem? Master's thesis, School of Computer Science, University of Magdeburg

Aldrich J (2005) Open modules: Modular reasoning about advice. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 3586. Springer, pp 144–168

Allan C, Avgustinov P, Christensen A, Hendren L, Kuzins S, Lhotak O, de Moor O, Sereni D, Sittampalam G, Tibble J (2005) Adding trace matching with free variables to AspectJ. In: Proc. Int'l Conf. Object-Oriented Programming, systems, languages, and applications (OOPSLA), ACM Press, pp 345–364

Alves V, Gheyi R, Massoni T, Kulesza U, Borba P, Lucena C (2006) Refactoring product lines. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 201–210

Anastasopoules M, Gacek C (2001) Implementing product line variabilities. In: Proc. Symposium on Software Reusability (SSR), ACM Press, pp 109–117

Ancona D, Damiani F, Drossopoulou S, Zucca E (2005) Polymorphic bytecode: Compositional compilation for Java-like languages. In: Proc. Int'l Symp. Principles of Programming Languages (POPL). ACM Press, pp 26–37

Ancona D, Zucca E (2001) True modules for Java-like languages. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 2072. Springer, pp 354–380

Apel S (2007) The role of features and aspects in software development. Ph.D. thesis, School of Computer Science, University of Magdeburg

Apel S (2010) How AspectJ is used: An analysis of eleven AspectJ programs. J Object Technol (JOT) 9(1):117–142

Apel S, Hutchins D (2010) A calculus for uniform feature composition. ACM Trans Program Lang Syst (TOPLAS) 32(5):1–33

Apel S, Kästner C (2009) An overview of feature-oriented software development. J Object Techno (JOT) 8(5):49–84

Apel S, Kästner C, Batory D (2008) Program refactoring using functional aspects. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 161–170

Apel S, Kästner C, Leich T, Saake G (2007) Aspect refinement—unifying AOP and stepwise refinement. J Object Technol (JOT)—Special Issue: TOOLS, EUROPE 2007 6(9):13–33

Apel S, Kästner C, Lengauer C (2009) FeatureHouse: Language-independent, automated software composition. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Computer Society, pp 221–231

Apel S, Kästner C, Lengauer C (2013a) Language-independent and automated software composition: The FeatureHouse experience. IEEE Trans Software Eng (TSE) 39(1):63–79

Apel S, Kolesnikov S, Liebig J, Kästner C, Kuhlemann M, Leich T (2012a) Access control in feature-oriented programming. Sci Comput Program (SCP) 77(3):174–187

Apel S, Leich T, Rosenmüller M, Saake G (2005) FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Lecture Notes in Computer Science, vol 3676. Springer, pp 125–140

Apel S, Leich T, Saake G (2008b) Aspectual feature modules. IEEE Trans Software Eng (TSE) 34(2):162–180

Apel S, Lengauer C, Möller B, Kästner C (2010) An algebraic foundation for automatic feature-based program synthesis. Sci Comput Program 75(11):1022–1047

Apel S, Leßenich O, Lengauer C (2012). Structured merge with auto-tuning: Balancing precision and performance. In: Proc. Int'l Conf. Automated Software Engineering (ASE). ACM Press, pp 120–129

Apel S, Liebig J, Brandl B, Lengauer C, Kästner C (2011) Semistructured merge: Rethinking merge in revision control systems. In: Proc. Europ. Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE). ACM Press, pp 190–200

Apel S, von Rhein A, Wendler P, Größlinger A, Beyer D (2013b) Strategies for product-line verification: Case studies and experiments. In: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE). IEEE Computer Society, pp 482–491

Appeltauer M, Hirschfeld R, Haupt M, Masuhara H (2011) ContextJ: Context-oriented programming with Java. Comput Softw 28(1):272–292

Aracic I, Gasiunas V, Mezini 1082 M, Ostermann K (2006) An overview of CaesarJ. Trans aspect-orient Softw Dev (TAOSD) 1(1):135–173

Arendt T, Biermann E, Jurack S, Krause C, Taentzer G (2010) Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS), lecture notes in computer science, vol. 6394. Springer, pp 121–135

Arnoldus J, Bijpost J, van den Brand M (2007) Repleo: A syntax-safe template engine. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 25–32

Atkins DL (1998) Version sensitive editing: Change history as a programming tool. In: Proc. ECOOP Symposium on System Configuration Management (SCM). Lecture Notes in Computer Science, vol 1439. Springer, pp 146–157

Atkins DL, Ball T, Graves TL, Mockus A (2002) Using version control data to evaluate the impact of software tools: A case study of the version editor. IEEE Trans Software Eng 28(7):625–637

Bass L, Clements P, Kazman R (1998) Software architecture in practice. Wesley

Batory D (2005) Feature models, grammars, and propositional formulas. In: Proc. Int'l Software Product Line Conference (SPLC), Lecture Notes in Computer Science, vol 3714. Springer, pp 7–20

Batory D, Höfner P, Kim J (2011) Feature interactions, products, and composition. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 13–22

Batory D, O'Malley S (1992) The design and implementation of hierarchical software systems with reusable components. ACM Trans Software Eng Methodol (TOSEM) 1(4):355–398

Batory D, Sarvela JN, Rauschmayer A (2004) Scaling step-wise refinement. IEEE Trans Software Eng (TSE) 30(6):355–371

Baxter I, Mehlich M (2001) Preprocessor conditional removal by simple partial evaluation. In: Proc. Working Conf. Reverse Engineering (WCRE). IEEE Computer Society, pp 281–290

Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntaxtrees. In: Proc. Int'l Conf. Software Maintenance (ICSM), IEEE Computer Society, pp 368–377

Benavides D, Seguraa S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. Inf Syst 35(6):615–636

Benavides D, Trinidad P, Ruiz-Cortes A (2005) Automated reasoning on feature models. In: Proc. Conf. Advanced Information Systems Engineering (CAiSE). Lecture notes in computer science, vol 3520. Springer, pp 491–503

Bergel A, Ducasse S, Nierstrasz O (2005) Classbox/J: Controlling the scope of change in Java. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 177–189

Berger T, She S, Czarnecki K, Wąsowski A (2010a) Feature-to-code mapping in two large product lines. In: Proc. Int'l Software Product Line Conference (SPLC). Lecture Notes in Computer Science, vol 6287. Springer, pp 498–499

Berger T, She S, Lotufo R, Wąsowski A, Czarnecki K (2010b) Variability modeling in the real: A perspective from the operating systems domain. In: Proc. Int'l Conf. Automated Software Engineering (ASE). ACM Press, pp 73–82

Berre DL, Parrain A (2010) The Sat4j library, release 2.2. J Satisf Boolean Model Comput (JSAT) 7(2–3):59–64

Beuche D, Papajewski H, Schröder-Preikschat W (2004) Variability management with featuremodels. Sci Comput Program 53(3):333–352

Biggerstaff T (1994) The library scaling problem and the limits of concrete component reuse. In: Proc. Int'l Conf. Software Reuse (ICSR), IEEE Computer Society, pp 102–109

Biggerstaff T, Mitbander BG, Webster D (1993) The concept assignment problem in program understanding. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 482–498

Binkley D et al (2006) Tool-supported refactoring of existing object-oriented code into aspects. IEEE Trans Softw Eng (TSE) 32(9):698–717

Blom J, Jonsson B, Kempe L (1994) Using temporal logic for modular specification of telephone services. In: Bouma LG, Velthuijsen H (eds) Feature interactions in telecommunications systems. IOS Press, pp 197–216

Blume M, Appel AW (1999) Hierarchical modularity. ACM Trans Program Lang Syst (TOPLAS) 21(4):813–847

Bodden E (2012) Inter-procedural data-flow analysis with IFDS/IDE and Soot. In: Int'l Workshop on the State of the Art in Java Program Analysis (SOAP). ACM Press, pp 3–8

Boehm BW (1988) A Spiral Model of Software Development and Enhancement. Computer 21(5):61–72. http://dx.doi.org/10.1109/2.59

Borba P, Teixeira L, Gheyi R (2010) A theory of software product line refinement. In: Proc. Int'l Colloquium Theoretical Aspects of Computing (ICTAC). Springer, pp 15–43

Bosch J (2000) Design and use of software architectures: Adopting and evolving a product-line approach. ACM Press/Addison-Wesley

Bosch J (2009) From software product lines to software ecosystems. In: Proc. Int'l Software Product Line Conference (SPLC), ACM Press, pp 111–119

Boucher Q, Classen A, Heymans P, Bourdoux A, Demonceau L (2010) Tag and prune: A pragmatic approach to software product line implementation. In: Proc. Int'l Conf. Automated Software Engineering (ASE). ACM Press, pp 333–336

Bowen T, Dworack F, Chow C, Griffeth N, Lin GHY-J (1989) The feature interaction problem in telecommunications systems. In: Proc. Int'l Conf. Software Engineering for Telecommunication Switching Systems (SETSS), IEEE Computer Society, pp 59–62

Boxleitner S, Apel S, Kästner C (2009) Language-independent quantification and weaving for feature composition. In: Proc. Int'l Symp. Software Composition (SC). Lecture Notes in Computer Science, vol 5634. Springer, pp 45–54

Brabrand C, Ribeiro M, Tolêdo T, Borba P (2012) Intraprocedural dataflow analysis for software product lines. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 13–24

BrabrandC, Schwartzbach MI (2002) Growing languages with metamorphic syntax macros. In: Proc. Int'l Symp. Partial Evaluation and Semantics-Based Program Manipulation (PEPM). ACM Press, pp 31–40

Bracha G, Cook W (1990) Mixin-based inheritance. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 303–311

Bruns G, Mataga P, Sutherland I (1998) Features as service transformers. In: Feature Interactions in Telecommunications Systems V. IOS Press, pp 85–97

Calder M, Kolberg M, Magill EH, Reiff-Marganiec S (2003) Feature interaction: A critical review and considered forecast. Comput Netw 41(1):115–141

Cardelli L (1997) Program fragments, linking, and modularization. In: Proc. Int'l Symp. Principles of Programming Languages (POPL). ACM Press, pp 266–277

Chacon S (2009) Pro Git. Apress. http://progit.org/

Chen K, Zhang W, Zhao H, Mei H (2005) An approach to constructing feature models based on requirements clustering. In: Proc. Int'l Conf. Requirements Engineering (RE). IEEE Computer Society, pp 31–40

Cheng B, de Lemos R, Giese H, Inverardi P, Magee J et al (2009) Software engineering for selfadaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol 5525. Springer, pp 1–26

Chu-Carroll M, Wright J, Ying A (2003) Visual separation of concerns through multidimensional program storage. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 188–197

Classen A, Heymans P, Schobbens P (2008) What's in a feature: A requirements engineering perspective. In: Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE) Lecture notes in computer science, vol 4961. Springer, pp 16–30

Classen A, Heymans P, Schobbens P-Y, Legay A, Raskin J-F (2010) Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 335–344

Classen A, Cordy M, Heymans P, Legay A, Schobbens P-Y (2012) Model checking software product lines with SNIP. Software Tools Technol Transfer (STTT) 14(5):589–612

Clements P, Krueger CW (2002) Point/counterpoint: Being proactive pays off/eliminating the adoption barrier. IEEE Softw 19(4):28–31

Clements P, Northrop L (2001) Software product lines: Practices and patterns. Addison-Wesle

Cohen MB, Dwyer MB, Shi J (2007) Interaction testing of highly-configurable systems in the presence of constraints. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA). ACM Press, pp 129–139

Cole L, Borba P (2005) Deriving refactorings for AspectJ. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 123–134

Colyer A, Clement A, Harley G, Webster M (2004a) Eclipse AspectJ: Aspect-oriented programming with AspectJ and the eclipse AspectJ development tools, 1st edn. Addison-Wesley Professional, Reading

Colyer A, Greenfield J, Jacobson I, Kiczales G, Thomas D (2005) Aspects: Passing fad or new foundation? In: Companion Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM Press, pp 376–377

Colyer A, Rashid A, Blair G (2004b) On the separation of concerns in program families. Technical Report COMP-001-2004, Computing Department, Lancaster University

Colyer A, Clement A (2004) Large-scale AOSD for middleware. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD), ACM Press, pp 56–65

Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R (2009) A systematic survey of program comprehension through dynamic analysis. IEEE Trans Software Eng 35(5):684–702

Czarnecki K, Antkiewicz M (2005) Mapping features to models: A template approach based on superimposed variants. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Lecture notes in computer science, vol 3676. Springer, pp 422–437

Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A (2012) Cool features and tough decisions: A comparison of variability modeling approaches. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 173–182

Czarnecki K, Helsen S, Eisenecker U (2005a) Formalizing cardinality-based feature models and their specialization. Softw Process: Improv Pract 10(1):7–29

Czarnecki K, Helsen S, Eisenecker U (2005b) Staged configuration through specialization and multilevel configuration of feature models. Softw Process: Improv Pract 10(2):143–169

Czarnecki K, Pietroszek K (2006) Verifying feature-based model templates against wellformedness OCL constraints. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 211–220

Czarnecki K, Eisenecker U (2000) Generative programming: Methods, tools, and applications. ACM Press/Addison-Wesley

da Mota Silveira Neto PA, do Carmo Machado I, McGregor JD, de Almeida ES, de Lemos Meira SR (2011) A systematic mapping study of software product lines testing. Inf Softw Technol 53(5):407–423

Dantas D, Walker D (2006) Harmless advice. In: Proc. Int'l Symp. Principles of Programming Languages (POPL), ACM Press, pp 383–396

Delaware B, Cook WR, Batory D (2009) Fitting the pieces together: A machine-checked model of safe composition. In: Proc. Europ. Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE). ACM Press, pp 243–252

DeRemer F, Kron HH (1976) Programming-in-the-large versus programming-in-the-small. IEEE Trans Softw Eng 2:80–86

Dessi M (2009) Spring 2.5 Aspect Oriented programming. Packt Publishing

Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D (2012a) A robust approach for variability extraction from the Linux build system. In: Proc. Int'l Software Product Line Conference (SPLC). ACM Press, pp 21–30

Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D (2012b) Understanding Linux feature distribution. In: Proc. AOSD Workshop on Modularity In Systems Software (MISS). ACM Press, pp 15–20

Dijkstra EW (1976) A discipline of programming. Prentice-Hall

Engström E, Runeson P (2011) Software product line testing—a systematic mapping study. Inf Softw Technol (IST) 53(1):2–13

Erl T (2005) Service-oriented architecture: Concepts, technology, and design. Prentice Hall

Ernst M, Badros G, Notkin D (2002) An empirical analysis of C preprocessor use. IEEE Trans Softw Eng (TSE) 28(12):1146–1170

Estler H-C, Ruhroth T, Wehrheim H (2007) Model checking correctness of refactorings—some experiments. Electron Notes Theor Comput Sci 187:3–17

Favre J-M (1995) The CPP paradox. In: Proc. European Workshop on Software Maintenance

Favre J-M (1997) Understanding-in-the-large. In: Proc. Int'l Workshop on Program Comprehension. IEEE Computer Society, p 29

Favre J-M (2003) CPP denotational semantics. In: Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM). IEEE Computer Society, pp 22–31

Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2012) Do background colors improve program comprehension in the #ifdef hell? Empirical Software Eng. Online first. doi:10.1007/s10664-012-9208-x

Felty A, Namjoshi K (2003) Feature specification and automated conflict detection. ACM Trans Software Eng Methodol (TOSEM) 12(1):3–27

Fernandez-Amoros D, Gil RH, Somolinos JC (2009) Inferring information from feature diagrams to product line economic models. In: Proc. Int'l Software Product Line Conference (SPLC). ACM Press, pp 41–50

Filman R, Friedman D (2005) Aspect-oriented programming is quantification and obliviousness. In: Aspect-Oriented Software Development, Addison-Wesley, pp 21–35

Filman RE, Elrad T, Clarke S, Aksit M (eds) (2005a) Aspect-oriented software development. Addison-Wesley

Flatt M, Krishnamurthi S, Felleisen M (1998) Classes and mixins. In: Proc. Int'l Symp. Principles of Programming Languages (POPL), ACM Press, pp 171–183

Ford H, Crowther S (1922) My life and work (the autobiography of Henry ford). Doubleday

Fowler M (1999) Refactoring: Improving the design of existing code. Addison-Wesley

Gamma E, Beck K (2003) Contributing to eclipse: Principles, patterns, and plugins. Wesley

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: Elements of reusable object oriented software. Addison-Wesley, Reading

Garlan D, Allen R, Ockerbloom J (1995) Architectural mismatch or why it's hard to build systems out of existing parts. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Computer Society, pp 179–185

Garvin BJ, Cohen MB (2011) Feature interaction faults revisited: An exploratory study. In: Proc. Int'l Symp. Software Reliability Engineering (ISSRE), IEEE Computer Society, pp 90–99

Gazzillo P, Grimm R (2012) SuperC: Parsing all of C by taming the preprocessor. In: Proc. Int'l Conf. Programming Language Design and Implementation (PLDI). ACM Press, pp 323–334

Gosling J, Joy B, Steele G, Bracha G (2005) Java^{TM} language specification. The Java^{TM} series, 3rdedn. Addison-Wesley, Reading

Gradecki JD, Lesiecki N (2003) Mastering AspectJ: Aspect-oriented programming in Java. John Wiley & Sons, Inc.

Griffeth N, Velthuijsen H (1994) The negotiating agents approach to runtime feature interaction resolution. In: Bouma LG, Velthuijsen H (eds) Feature interactions in telecommunications systems. IOS Press, pp 217–235

Griss M (2000) Implementing product-line features by composing aspects. In: Proc. Int'l Software Product Line Conference (SPLC). Kluwer Academic Publishers, pp 271–288

Griss ML, Favaro J, d' Alessandro M (1998) Integrating feature modeling with the RSEB. In: Proc. Int'l Conf. Software Reuse (ICSR). IEEE Computer Society, p 76

Haase S (2012) A program slicing approach to feature identification. Master's thesis, School of Computer Science, University of Magdeburg

Hall RJ (2005) Fundamental nonmodularity in electronic mail. Autom Software Eng 12(1):41–79

Hanenberg S, Oberschulte C, Unland R (2003) Refactoring of aspect-oriented software. In: Proc. Int'l Conf. Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), pp 19–35 (tranSIT GmbH)

Harbulot B, Gurd J (2006) A join point for loops in AspectJ. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD), ACM Press, pp 63–74

Hay J, Atlee J (2000) Composing features and resolving interactions. In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), ACM Press, pp 110–119

Heidenreich F, Kopcsek J, Wende C (2008b) FeatureMapper: Mapping features to models. In: Companion Int'l Conf. Software Engineering (ICSE). ACM Press, pp 943–944

Heidenreich F, Şavga I, Wende C (2008a) On controlled visualisations in software product line engineering. In: Proc. SPLCWorkshop on Visualization in Software Product Line Engineering (ViSPLE) Lero, pp 303–313

Heidenreich F, Sánchez P, Santos J a, Zschaler S, Alférez M, Araújo J. a, Fuentes L, Kulesza U, Moreira A, Rashid A (2010) Relating feature models to other models of a software product line: A comparative study of FeatureMapper and VML. In: Transactions on aspect-oriented software development VII. Springer, pp 69–114

Herrmann S (2002) Object teams: Improving modularity for crosscutting collaborations. In: Proc. Int'l Conf. Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays). Lecture Notes in Computer Science, vol 2591. Springer, pp 248–264

Heymans P (2012) Formal methods for the masses. In: Proc. Int'l Software Product Line Conference (SPLC), ACM Press, p 4

Hirschfeld R, Costanza P, Nierstrasz O (2008) Context-oriented programming. J Object Technol (JOT) 7(3):125–151

Hofer W, Elsner C, Blendinger F, Schröder-Preikschat W, Lohmann D (2011) Tool chain independent variant management with the Leviathan filesystem. In: Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD) ACM Press, pp 18–24

Hu Y, Merlo E, Dagenais M, Laguë B (2000) C/C++ conditional compilation analysis using symbolic execution. In: Proc. Int'l Conf. Software Maintenance (ICSM). IEEE Computer Society, pp 196–206

Huang SS, Smaragdakis Y (2011) Morphing: Structurally shaping a class by reflecting on others. ACM Trans Prog Lang Syst (TOPLAS) 33(2), 6:1–6:44

Huang SS, Zook D, Smaragdakis Y (2005) Statically safe program generation with SafeGen. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Lecture Notes in Computer Science, vol 3676. Springer, pp 309–326

Hubaux A, Xiong Y, Czarnecki K (2012) A user survey of configuration challenges in Linux and eCos. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 149–155

Hunleth F, Cytron RK (2002) Footprint and feature management using aspect-oriented programming techniques. In: Proc. Conf. Languages, Compilers and Tools For Embedded systems (LCTES), ACM Press, pp 38–45

Hunleth F, Cytron RK (2002) Footprint and feature management using aspect-oriented programming techniques. In: Proc. Conf. Languages, Compilers and Tools For Embedded Systems (LCTES). ACM Press, pp 38–45

Jackson M, Zave P (1998) Distributed feature composition: A virtual architecture for telecommunications services. IEEE Trans Software Eng (TSE) 24(10):831–847

Janota M (2010) SAT solving in interactive configuration. Ph.D. thesis, Department of Computer Science, University College Dublin

Janzen D, De Volder K. (2004). Programming with crosscutting effective views. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 3086. Springer, pp 195–218

Jarzabek S, Bassett P, Zhang H, Zhang W (2003) XVCL: XML-based variant configuration language. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 810–811

Johnson RE, Foote B (1988) Designing reusable classes. J Object-Oriented Program 1(2):22–35

Jones ND, Gomard CK, Sestoft P (1993) Partial evaluation and automatic program generation. Prentice-Hall

Kang K, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech Rep CMU/SEI-90-TR-21, SEI

Kang K, Kim S, Lee J, Kim K, Kim G, Shin E (1998) FORM: A feature-oriented reuse method with domain-specific reference architectures. Ann Softw Eng 5(1):143–168

Kang K, Lee J, Donohoe P (2002) Feature-oriented project line engineering. IEEE Softw 19:58–65

Kang KC, Sugumaran V, Park S (eds) (2009) Applied software product line engineering. Auerbach Publications

Kästner C (2007) Aspect-oriented refactoring of Berkeley DB. Master's thesis, School of Computer Science, University of Magdeburg

Kästner C (2010) Virtual separation of concerns. Ph.D. thesis, School of Computer Science, University of Magdeburg

Kästner C, Apel S (2009) Virtual separation of concerns—a second chance for preprocessors. J Object Technol 8(6):59–78

Kästner C, Apel S, Batory D (2007) A case study implementing features using AspectJ. In: Proc. Int'l Software Product Line Conference (SPLC). IEEE Computer Society, pp 223–232

Kästner C, Apel S, Kuhlemann M (2008) Granularity in software product lines. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 311–320

Kästner C, Apel S, Kuhlemann M (2009) A model of refactoring physically and virtually separated features. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 157–166

Kästner C, Apel S, Ostermann K (2011) The road to feature modularity? In: Proc. SPLC Workshop on Feature-Oriented Software Development (FOSD), ACM Press, pp 5:1–5:8

Kästner C, Apel S, Thüm T, Saake G (2012a) Type checking annotation-based product lines. ACM Trans Softw Eng Methodol (TOSEM) 21(3):14:1–14:39

Kästner C, Apel S, Trujillo S, Kuhlemann M, Batory D (2009b) Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In: Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE). Lecture Notes in Business Information Processing, vol 33. Springer, pp 175–194

Kästner C, Apel S, ur Rahman SS, Rosenmüller M, Batory D, Saake G (2009) On the impact of the optional feature problem: Analysis and case studies. In: Proc. Int'l Software Product Line Conference (SPLC), ACM Press, pp 181–190

Kästner C, Giarrusso PG, Ostermann K (2011) Partial preprocessing of C code for variability analysis. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 137–140

Kästner C, Giarrusso PG, Rendel T, Erdweg S, Ostermann K, Berger T (2011) Variability-aware parsing in the presence of lexicalmacros and conditional compilation. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM Press, pp 805–824

Kästner C, Trujillo S, 432 Apel S (2008b) Visualizing software product line variabilities in source code. In: Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE). Lero, University of Limerick, pp 303–313

Kästner C, von Rhein A, Erdweg S, Pusch J, Apel S, Rendel T, Ostermann K (2012c) Toward variability-aware testing. In: Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD). ACM Press, pp 1–8

Kästner C, Ostermann K, Erdweg S (2012b). Avariability-awaremodule system. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM Press, pp 773–792

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W (2001) An overview of AspectJ. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 2072. Springer, pp 327–353

Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect oriented programming. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 1241. Springer, pp 220–242

Kim CHP, Khurshid S, Batory D (2012) Shared execution for efficiently testing product lines. In: Proc. Int'l Symp. Software Reliability Engineering (ISSRE). IEEE Computer Society pp 221–230

Kniesel G, Koch H (2004) Static composition of refactorings. Sci Comput Progr (SCP) 52(1–3):9–51

Kolberg M, Magill E, Marples D, Reiff S (2000) Results of the second feature interaction contest. In: Calder M, Magill E (eds) Feature interactions in telecommunication systems, vol VI. IOS Press, pp 311–325

Krone M, Snelting G (1994) On the inference of configuration structures from source code. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 49–57

Krueger CW (2002) Easing the transition to software mass customization. In: Proc. Int'l Workshop on Software Product-Family Engineering (PFE) Lecture Notes in Computer Science, vol 2290. Springer, pp 282–293

Krueger CW (2006) New methods in software product line practice. Commun ACM 49:37–40

Kuhlemann M, Batory D, Apel S (2009a) Refactoring feature modules. In: Proceedings of the International Conference on Software Reuse (ICSR), Springer, pp 106–115

KuhlemannM, Batory D, Kästner C (2009b) Safe composition of non-monotonic features. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), ACM Press, pp 177–185

Kuhn DR, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Software Eng (TSE) 30:418–421

Kühne T (1999) A functional pattern system for object-oriented design. Ph.D. thesis, Department of Computer Science, Darmstadt University of Technology

Laddad R (2003) AspectJ in action: Practical aspect-oriented programming. Manning Publications

Laddad R (2003b) AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications

Lafferty D, Cahill V (2003) Language-independent aspect-oriented programming. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 1–12

Lämmel R (1999) Declarative aspect-oriented programming. In: Proc. Int'l Symp. Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press, pp 131–146

Latendresse M (2003). Fast symbolic evaluation of C/C++ preprocessing using conditional values. In: Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, pp 170–179

Latendresse M (2004) Rewrite systems for symbolic evaluation of C-like preprocessing. In: Proc. Int'l Conf. Automated Software Engineering (CSMR). IEEE Computer Society, pp 165–173

Lauenroth K, Pohl K, Toehning S (2009) Model checking of domain artifacts in product line engineering. In: Proc. Int'l Conf. Automated Software Engineering (ASE), IEEE Computer Society, pp 269–280

Le D, Walkingshaw E, Erwig M (2011) #ifdef confirmed harmful: Promoting understandable software variation. In: Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC). IEEE Computer Society, pp 143–150

Lee K et al (2006) Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: Proc. Int'l Software Product Line Conference (SPLC), IEEE Computer Society, pp 103–112

Leich T (2012) Variables Nanodatenmanagement für eingebettete Systeme. Ph.D. thesis, School of Computer Science, University of Magdeburg

Leich T, Apel S, Rosenmüller M, Saake G (2005) Handling optional features in software product lines. In: OOPSLA workshop on managing variabilities consistently in design and code. http://www.kircher-schwanninger.de/workshops/MVCDC/

Leroy X (1994) Manifest types, modules, and separate compilation. In: Proc. Int'l Symp. Principles of Programming Languages (POPL). ACM Press, pp 109–122

Li HC, Krishnamurthi S, Fisler K (2005) Modular verification of open features using three-valued model checking. Autom Softw Eng 12(3):349–382

Lieberherr KJ, Lorenz DH, Ovlinger J (2003) Aspectual collaborations—Combining modules and aspects. Comput J 46(5):542–565

Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 105–114

Liebig J, Kästner C, Apel S (2011) Analyzing the discipline of preprocessor annotations in 30million lines of C code. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 191–202

Liebig J, von Rhein A, Kästner C, Apel S, Dörre J, Lengauer C (2013) Scalable analysis of variable software. In: Proc. Europ. Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE). ACM Press, to appear

Lin F, Lin Y-J (1994) A building block approach to detecting and resolving feature interactions. In: Bouma LG, Velthuijsen H (eds) Feature interactions in telecommunications systems. IOS Press, pp 86–119

Liu J, Batory D, Lengauer C (2006) Feature oriented refactoring of legacy applications. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press, pp 112–121

Lohmann D, Hofer W, Schröder-Preikschat W, Spinczyk O (2011) Aspect-aware operating system development. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD), ACM Press, pp 69–80

Lohmann D, Scheler F, Tartler R, Spinczyk O, Schröder-Preikschat W (2006a) A quantitative analysis of aspects in the eCos kernel. In: Proc. Int'l EuroSys Conference (EuroSys). ACM Press, pp 191–204

Lohmann D, Spinczyk O, Schröder-Preikschat W (2006b) Lean and efficient system software product lines: Where aspects beat objects. Trans Aspect-Orient Softw Dev (TAOSD) 2(1):227–255

Lopez-Herrejon R (2006) Understanding feature modularity. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin

Lopez-Herrejon R, Batory D, Cook W (2005) Evaluating support for features in advanced modularization technologies. In: Proc. Int'l Conf. Generative and Component-Based Software Engineering (ECOOP). Lecture notes in computer science, vol 3586. Springer, pp 169–194

Lotufo R, She S, Berger T, Czarnecki K, Wąsowski A (2010) Evolution of the Linux kernel variability model. In: Proc. Int'l Software Product Line Conference (SPLC). Springer, pp 136–150

Madsen OL, Moller-Pedersen B (1989) Virtual classes: A powerful mechanism in object-oriented programming. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 397–406

Masuhara H, Kawauchi K (2003) Dataflow pointcut in aspect-oriented programming. In: Proc. Asian Symp. Programming Languages and Systems (APLAS), Lecture Notes in Computer Science, vol 2895. Springer, pp 105–121

McCloskey B, Brewer E (2005) ASTEC: A new approach to refactoring C. In: Proc. Europ. Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE). ACM Press, pp 21–30

McIlroy MD (1969) Mass produced software components. In: Software engineering: Report of a conference sponsored by the NATO science committee, Garmisch, Germany, 7–11 Oct. 1968, Scientific Affairs Division, NATO, pp 138–155

Mehner K, Rashid A (2003) Towards a generic model for AOP (GEMA). Technical report CSEG/1/03, Computing Department, Lancaster University

Mendonça M, Wąsowski A, Czarnecki K (2009) SAT-based analysis of feature models is easy. In: Proc. Int'l Software Product Line Conference (SPLC). ACM Press, pp 231–240

Mens T (2002) A state-of-the-art survey on software merging. IEEE Trans Softw Eng (TSE) 28(5):449–462

Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng (TSE) 30(2): 126–139

Metzger A, Pohl K, Heymans P, Schobbens P-Y, Saval G (2007) Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: Proc. Int'l Conf. Requirements Engineering (RE). IEEE Computer Society, pp 243–253

Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall

Mezini M, Ostermann K (2004) Variability management with feature-oriented programming and aspects. In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), ACM Press, pp 127–136

Michel R, Classen A, Hubaux A, Boucher Q (2011) A formal semantics for feature cardinalities in feature diagrams. In: Proc. Int'l Workshop on Variability Modelling of Softwareintensive Systems (VaMoS). ACM Press, pp 82–89

Monteiro MP, Fernandes JM (2005) Towards a catalog of aspect-oriented refactorings. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 111–122

Murphy GC, Lai A, Walker R, Robillard M (2001) Separating features in source code: An exploratory study. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 275–284

Murphy-Hill E, Black AP (2008) Breaking the barriers to successful refactoring: Observations and tools for extract method. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 421–430

Muthig D, Patzke T (2002) Generic implementation of product line components. In: Proc. Int'l Conf. Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), Lecture Notes in Computer Science, vol 2591. Springer, pp 313–329

Nadi S, Holt RC (2012) Mining Kbuild to detect variability anomalies in Linux. In: Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, pp 107–116

Neves L, Teixeira L, Sena D, Alves V, Kulezsa U, Borba P (2011) Investigating the safe evolution of software product lines. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 33–42

Nhlabatsi A, Laney R, Nuseibeh B (2008) Feature interaction: The security threat from within software systems. Prog inform 5:75–89

Opdyke WF (1992) Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign

Oster S, Markert 1235 F, Ritter P (2010) Automated incremental pairwise testing of software product lines. In: Proc. Int'l Software Product Line Conference (SPLC). Lecture Notes in Computer Science, vol 6287. Springer, pp 196–210

Ostermann K, Giarrusso PG, Kästner C, Rendel T (2011) Revisiting information hiding: Reflections on classical and nonclassical modularity. In: Proc. Europ. Conf. Object- Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 6813. Springer, pp 155–178

Ostermann K, Mezini M, Bockisch C (2005) Expressive pointcuts for increased modularity. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 3586. Springer, pp 214–240

Ouellet M, Merlo E, Sozen N, Gagnon M (2012) Locating features in dynamically configured avionics software. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 1453–1454

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

Parnas DL (1979) Designing software for ease of extension and contraction. IEEE Trans Software Eng (TSE), SE-5(2):128–138

Parnas, DL (1976) On the design and development of program families. IEEE Trans Software Eng (TSE), 2(1):1–9

Parr TJ (2004) Enforcing strict model-view separation in template engines. In: Proc. Int'l Conference on World Wide Web. ACM Press, pp 224–233

Perrouin G, Sen S, Klein J, Baudry B, le Traon Y (2010) Automated and scalable t-wise test case generation strategies for software product lines. In: Proc. Int'l Conf. Software Testing, Verification, and Validation. IEEE Computer Society, pp 459–468

Pierce BC (2002) Types and programming languages. MIT Press

Pohl K, Böckle G, van der Linden FJ (2005) Software product Line engineering: Foundations, principles and techniques. Springer

Pomakis K, Atlee J (1996) Reachability analysis of feature interactions: A progress report. In: Proc. Int'l Symp. Software Testing and Analysis (ISSTA), ACM Press, pp 216–223

Popovici A, Alonso G, Gross T (2003) Just-in-time aspects: Efficient dynamic weaving for Java. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD), ACM Press, pp 100–109

Poshyvanyk D, Guéhéneuc Y-G, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEETrans Software Eng 33(6):420–432

Post H, Sinz C (2008) Configuration lifting: Verification meets software configuration. In: Proc. Int'l Conf. Automated Software Engineering (ASE). IEEE Computer Society, pp 347–350

Prehofer C (1997) Feature-oriented programming: A fresh look at objects. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol 1241. Springer, pp 419–443

Rabkin A, Katz R (2011) Static extraction of program configuration options. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Computer Society, pp 131–140

Rashid A, Royer J-C, Rummler A (2011) Aspect-oriented, model-driven software product lines: The AMPLE way. Cambridge University Press

Reenskaug T, Andersen E, Berre A, Hurlen A, Landmark A, Lehne O, Nordhagen E, Ness-Ulseth E, Oftedal G, Skaar A, Stenslet P (1992) OORASS: Seamless support for the creation and maintenance of object-oriented systems. J Object-Orient Program (JOOP) 5(6):27–41

Refstrup JG (2009) Adapting to change: Architecture, processes and tools: A closer look at HP's experience in evolving the Owen software product line. In: Proc. Int'l Software Product Line Conference (SPLC). Keynote presentation

Reisner E, Song C, Ma K-K, Foster JS, Porter A (2010) Using symbolic evaluation to understand behavior in configurable software systems. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press, pp 445–454

Ribeiro M, Pacheco H, Teixeira L, Borba P (2010) Emergent feature modularization. In: Companion Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM Press, pp 11–18

Robillard M, Weigand-Warr F (2005) ConcernMapper: Simple view-based separation of scattered concerns. In: Proc. OOPSLA Workshop on Eclipse Technology eXchange (ETX). ACM Press, pp 65–69

Robillard M, Murphy GC (2002) Concern graphs: Finding and describing concerns using structural program dependencies. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 406–416

Rosenmüller M, Apel S, Leich T, Saake G (2009a) Tailor-made data management for embedded systems: A case study on Berkeley DB. Data Knowl Eng (DKE) 68(12):1493–1512

Rosenmüller M, Kästner C, Siegmund N, Sunkle S, Apel S, Leich T, Saake G (2009b) SQL à la carte—toward tailor-made data management. In: Proc. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW) Lecture Notes in Informatics. Gesellschaft für Informatik (GI), vol P-144, pp 117–136

RosenmüllerM(2011) Towards flexible feature composition: Static and dynamic binding in software product lines. Ph.D. thesis, School of Computer Science, University of Magdeburg

Rosenmüller M, Siegmund N, Apel S, Saake G (2011) Flexible feature binding in software product lines. Autom Software Eng 18(2):163–197

Rosenmüller M, Siegmund N, Schirmeier H, Sincero J, Apel S, Leich T, Spinczyk O, Saake G (2008) FAME-DBMS: Tailor-made data management solutions for embedded systems. In: Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management. ACM Press, pp 1–6

Saake G, Rosenmüller M, Siegmund N, Kästner C, Leich T (2009) Downsizing data management for embedded systems. Egyptian Comput Sci J 31(1):1–13

Sato Y, Chiba S, Tatsubori M (2003) A selective, just-in-time aspect weaver. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). Lecture Notes in Computer Science, vol 2830. Springer, pp 189–208

Savolainen J, Bosch J, Kuusela J, Männistö T (2009) Default values for improved product line management. In: Proc. Int'l Software Product Line Conference (SPLC). Carnegie Mellon University, pp 51–60

Schaefer I, Bettini L, Damiani F (2011) Compositional type-checking for delta-oriented programming. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 43–56

Schaefer I, Bettini L, Damiani F, Tanzarella N (2010) Delta-oriented programming of software product lines. In: Proc. Int'l Software Product Line Conference (SPLC), Springer, pp 77–91

Schmid K, Rabiser R, Grünbacher P (2011) A comparison of decision modeling approaches in product lines. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 119–126

Schobbens P-Y, Heymans P, Trigaux J-C, Bontemps Y (2007) Generic semantics of feature diagrams. Comput Netw 51(2):456–479

Schulze S (2013) Analysis and removal of code clones in software product lines. Ph.D. thesis, School of Computer Science, University of Magdeburg

Schulze S, Thüm T, Kuhlemann M, Saake G (2012) Variant-preserving refactoring in feature oriented software product lines. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM Press, pp 73–81

She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K (2011) Reverse engineering feature models. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, pp 461–470

She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K (2010) The variability model of the Linux kernel. In: Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS). University of Duisburg-Essen, pp 45–51

Siegmund N, Kästner C, Rosenmüller M, Heidenreich F, Apel S, Saake G (2009a) Bridging the gap between variability in client application and database schema. In: Proc. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW). Lecture Notes in Informatics, vol. P-144. Gesellschaft für Informatik (GI), pp 297–306

Siegmund N, Rosenmüller M, Kuhlemann M, Kästner C, Apel S, Saake G (2011) SPL Conqueror: Toward optimization of non-functional properties in software product lines. Softw Qual J Spec Issue Qual Eng Softw Prod Lines 3:487–517

Siegmund N, Rosenmüller M, Moritz G, Saake G, Timmermann D (2009b) Towards robust data storage in wireless sensor networks. IETE Tech Rev 26(5):335–340

Simos MA (1995) Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In: Proc. Symp. Software Reusability (SSR). ACM Press, pp 196–205

Sincero J, Schirmeier H, Schröder-Preikschat W, Spinczyk O (2007) Is the Linux kernel a software product line? In: Proc. Int'l Workshop Open Source Software and Product Lines (SPLC-OSSPL)

Sincero J, Schröder-Preikschat W, Spinczyk O (2010) Approaching non-functional properties of software product lines: Learning from products. In: Proc. Asia-Pacific Software Engineering Conf. (APSEC). IEEE Computer Society, pp 147–155

Singh N, Gibbs C, Coady Y (2007) C-CLR: A tool for navigating highly configurable system software. In: Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS). ACM Press, p 9

Smaragdakis Y, Batory D (2002a) Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans Softw Eng Methodol 11(2):215–255

Smaragdakis Y, Batory D (2002) Mixin layers: An Object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans Software Eng Methodol (TOSEM) 11(2):215–255

Snyder A (1986) Encapsulation and inheritance in object-oriented programming languages. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 38–45

Sommerville I (2010) Software engineering, 9th edn. Pearson Addison Wesley

Spencer H, Collyer G (1992) #ifdef considered harmful or portability experience with C news. In: Proc. USENIX Conf., USENIX Association, pp 185–198

Spinczyk O, Lohmann D, Urban M (2005) AspectC++: An AOP extension for C++. Softw Dev J 14:68–74

Spinczyk O (2002) Aspektorientierung und Programmfamilien im Betriebssystembau. Ph.D. thesis, School of Computer Science, University of Magdeburg

Staples M, Hill D (2004) Experiences adopting software product line development without a product line architecture. In: Proc. Asia-Pacific Software Engineering Conf. (APSEC). IEEE Computer Society, pp 176–183

Steimann F (2005) Domain models are aspect free. In: Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS). Lecture Notes in Computer Science, vol 3713. Springer, pp 171–185

Steimann F (2006) The paradoxical success of aspect-oriented programming. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 481–497

Steimann F, Pawlitzki T, Apel S, Kästner C (2010) Types and modularity for implicit invocation with implicit announcement. ACM Trans Software Eng Methodol (TOSEM) 20(1):1:1–1:43

Störzer M, Koppen C (2004) PCDiff: Attacking the fragile pointcut problem, abstract. In: European Interactive Workshop on Aspects in Software

Streitferdt D, Riebisch M, Philippow I (2003) Details of formalized relations in feature models using OCL. In: Proc. Int'l Conf. and Workshop Engineering of Computer-Based Systems (ECBS). IEEE Computer Society, pp 297–304

Strniša R, Sewell P, Parkinson M (2007) The Java module system: Core design and semantic definition. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM Press, pp 499–514

Sullivan K, Griswold W, Song Y, Cai Y, Shonle M, Tewari N, Rajan H (2005) Information hiding interfaces for aspect-oriented design. In: Proc. Int'l Symp. Foundations of Software Engineering (FSE), ACM Press, pp 166–175

Sultana N, Thompson S (2008) Mechanical verification of refactorings. In: Proc. Int'l Symp. Partial Evaluation and Semantics-Based Program Manipulation (PEPM). ACM Press, pp 51–60

Sunyé G, Pollet D, Traon YL, Jézéquel J-M (2001) Refactoring UML models. In: Proc. Int'l Conf. UML. Modeling Languages, Concepts, and Tools, of Lecture Notes in Computer Science, vol. 2185. Springer, pp 134–148

Svahnberg M, van Gurp J, Bosch J (2005) A taxonomy of variability realization techniques. SoftwPract Exp 35(8):705–754

Szewczyk R et al (2004) Habitat monitoring with sensor networks. Commun ACM 47(6):34–40

Szyperski C (1997) Component software: Beyond object-oriented programming. Wesley

Tamrawi A, Nguyen HA, Nguyen HV, Nguyen TN (2012) Build code analysis with symbolic evaluation. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 650–660

Tarr P, Ossher H, Harrison W, Sutton S Jr (1999) N degrees of separation: Multi-dimensionalseparation of concerns. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Computer Society, pp 107–119

Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W (2011) Feature consistency in compiletime-configurable system software: Facing the linux 10,000 feature problem. In: Proc. Int'l EuroSys Conference (EuroSys). ACM Press, pp 47–60

Tešanovic′ A, Sheng K, Hansson J (2004) Application-tailored database systems: A case of aspects in an embedded database. In: Proc. Int'l Database Engineering and Applications Symposium. IEEE Computer Society, pp 291–301

Thaker S, Batory D, Kitchin D, Cook W (2007) Safe composition of product lines. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 95–104

Thüm T, Apel S, Kästner C, Kuhlemann M, Schaefer I, Saake G (2012a) Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg

Thüm T, Batory D, Kästner C (2009) Reasoning about edits to feature models. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society, pp 254–264

Thüm T, Kästner C, Erdweg S, Siegmund N (2011a) Abstract features in feature modeling. In: Proc. Int'l Software Product Line Conference (SPLC). IEEE Computer Society, pp 191–200

Thüm T, Schaefer I, Apel S, Hentschel M (2012b) Family-based theorem proving for deductive verification of software product lines. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE). ACM Press, pp 11–20

Thüm T, Schaefer I, KuhlemannM, Apel S (2011b) Proof composition for deductive verification of software product lines. In: Proc. Int'l Workshop on Variability-Intensive Systems Testing, Validation & Verification (VAST). IEEE Computer Society, pp 270–277

Tsang S, Magill E (1998) Learning to detect and avoid run-time feature interactions in intelligent networks. IEEE Trans Software Eng (TSE) 24(10):818–830

Utas G (1998) A pattern language of feature interaction. In: Kimbler K, Bouma LG (eds) Feature interactions in telecommunications systems V, IOS Press, pp 98–114

van der Linden FJ, Schmid K, Rommes E (2007) Software product lines in action: The best industrial practice in product line engineering. Springer

van der Linden R (1994) Using an architecture to help beat feature interaction. In: Bouma W, Velthuijsen H (eds) Feature interactions in telecommunications systems. IOS Press, pp 24–35

van der Storm T (2004).Variability and component composition. In: Proc. Int'l Conf. Software Reuse (ICSR) Lecture notes in computer science, vol 3107. Springer, pp 157–166

van Ommering R (2002) Building product populations with software components. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press, pp 255–265

VanHilst M, Notkin D (1996) Using role components in implement collaboration-based designs. In: Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, pp 359–369

Voelter M, Groher I (2007) Product line implementation using Aspect-oriented and model-driven software development. In: Proc. Int'l Software Product Line Conference (SPLC), IEEE Computer Society, pp 233–242

Wand M, Kiczales G, Dutchyn C (2004) A semantics for advice and dynamic join points in aspect oriented programming. ACM Trans Program Lang Syst (TOPLAS) 26(5):890–910

Weise D, Crew R (1993) Programmable syntax macros. In: Proc. Int'l Conf. Programming Language Design and Implementation (PLDI). ACM Press, pp 156–165

Zave P (2003) An experiment in feature engineering. In: Programming Methodology. Springer, pp 353–377

Zave P (2010) Modularity in distributed feature composition. In: Nuseibeh B, Zave P (eds) Software requirements and design: The work of Michael Jackson. Good Friends Publishing, pp 267–290

Zhang C, Jacobsen H-A (2003) Quantifying aspects in middleware platforms. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). ACM Press, pp 130–139

Index