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):

DOI: 10.1007/978-3-642-37521-7_2, © Springer-Verlag Berlin Heidelberg 2013

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, `f` = (`R`, `W`, `S`), where `R` represents the requirements the feature satisfies, `W` the assumptions the feature takes about its environment and `S` its specification"
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

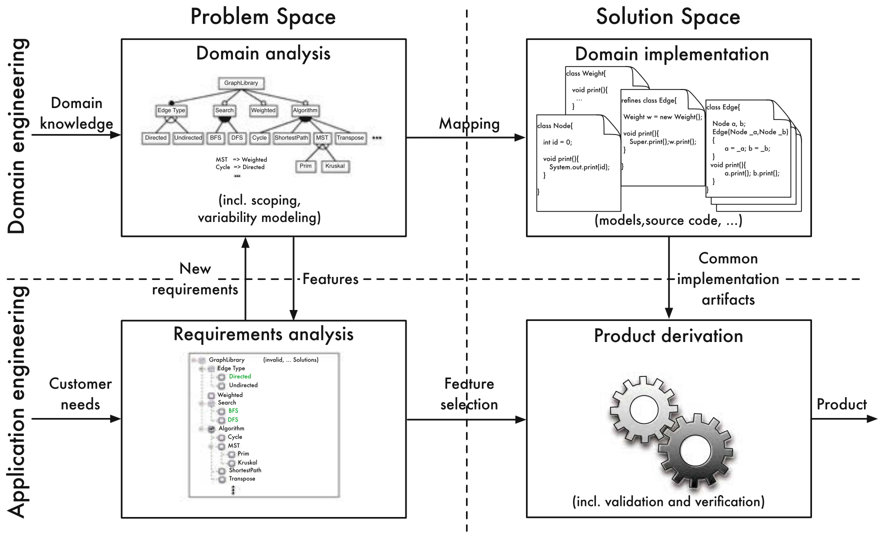
Most processes of traditional software engineering target the life cycle of a single software system. Independent of the specifics of the process used, developers collect requirements for the target system, and design and implement the system, either in separate, consecutive phases or in agile cycles. 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) □



```
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. 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 $\mathtt{f}_\mathtt{i}$ 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

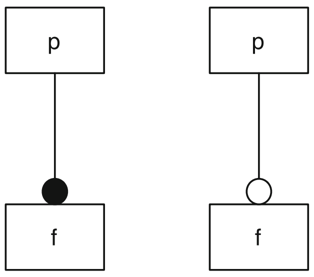


Fig. 2.2 Graphical notation for optional and mandatory features. A filled bullet denotes a mandatory feature, and an empty bullet denotes an optional feature

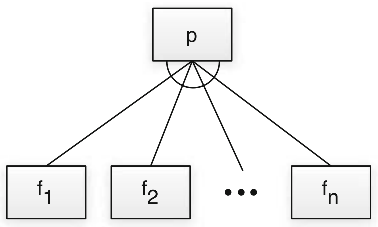


Fig. 2.3 Graphical notation for a one-out-of many choice. This choice corresponds to a generalized `xor` operator

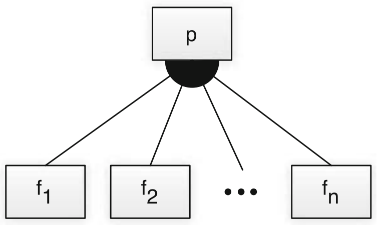


Fig. 2.4 Graphical notation for a some-out-of-many choice. This choice corresponds to the logical `or` operator

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.

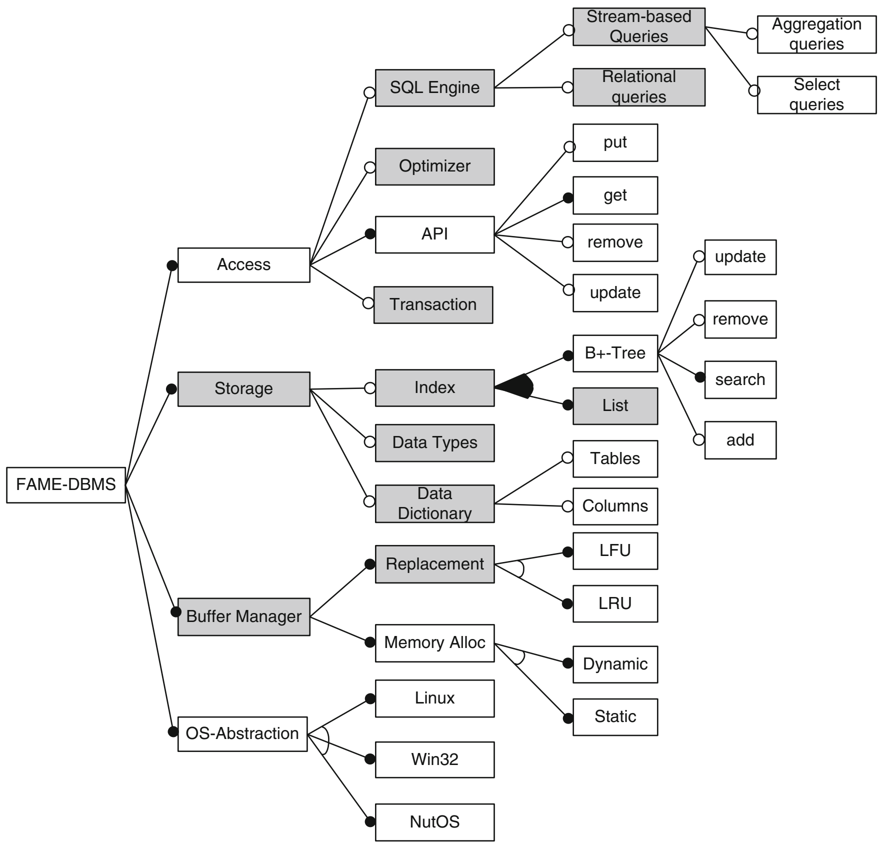


Fig. 2.5 Sample feature diagram for embedded data management

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:

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

□

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 $\mathtt{F}$. 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*.

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

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

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

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

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

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.

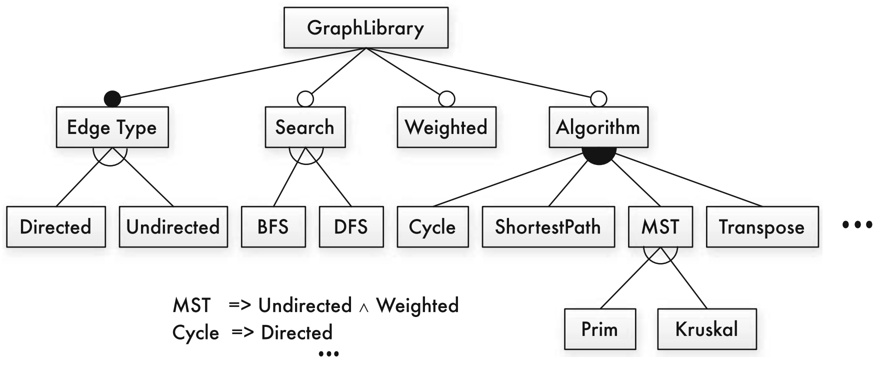


Fig. 2.6 A possible feature diagram of the graph library

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:

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

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:

$$\big(\texttt{OS} \Leftrightarrow (\texttt{Linux}\lor\texttt{Win}\lor\texttt{Mac})\big)\land\big(\neg(\texttt{Linux}\land\texttt{Win})\land\neg(\texttt{Linux}\land\texttt{Mac})\land\neg(\texttt{Win}\land\texttt{Mac})\big)$$

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

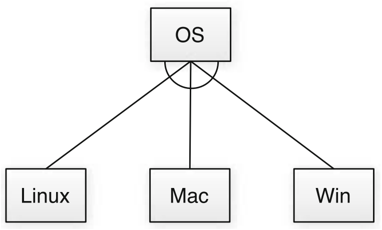


Fig. 2.7 Feature diagram for alternative operating systems

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.

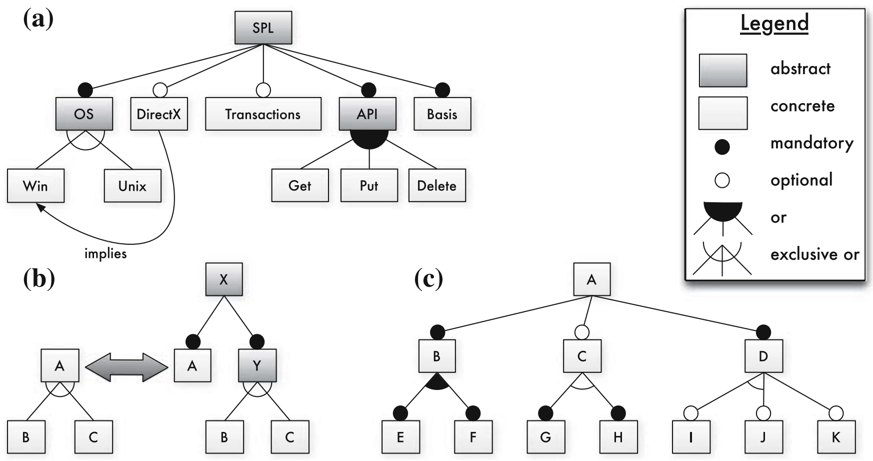


Fig. 2.8 Some variations of feature diagrams: **a** cross-tree constraints, **b** transformation toward abstract inner features, **c** mixing optionality and group constraints

[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).

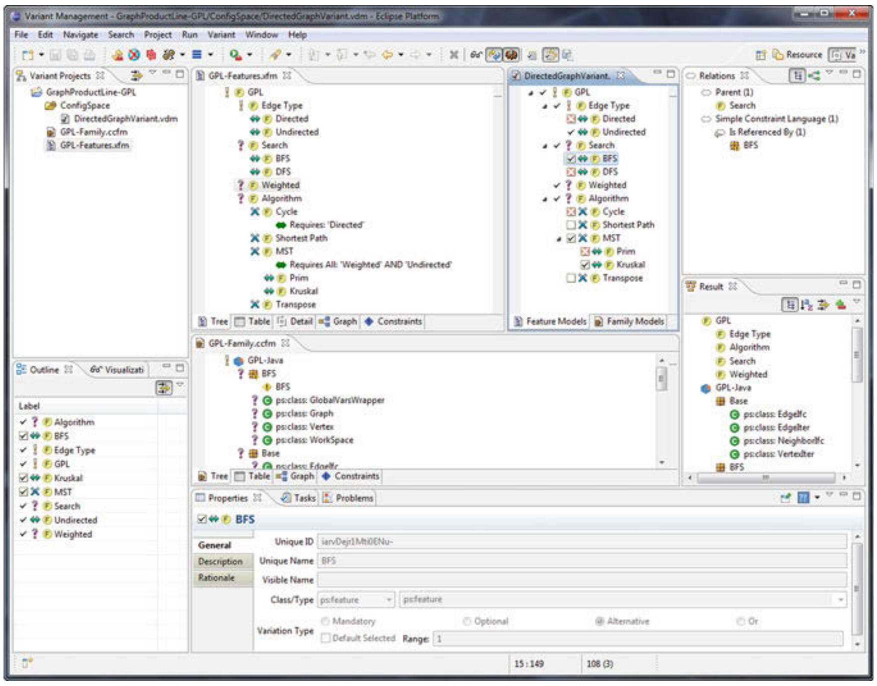


Fig. 2.10 Variability of the graph library modeled with pure::variants

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 $\texttt{SPL}_0$ which realizes an initial version of the envisioned software product line. In incremental steps from $\texttt{SPL}_\texttt{i}$ to $\texttt{SPL}_{\texttt{i}+1}$, 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.