*21*

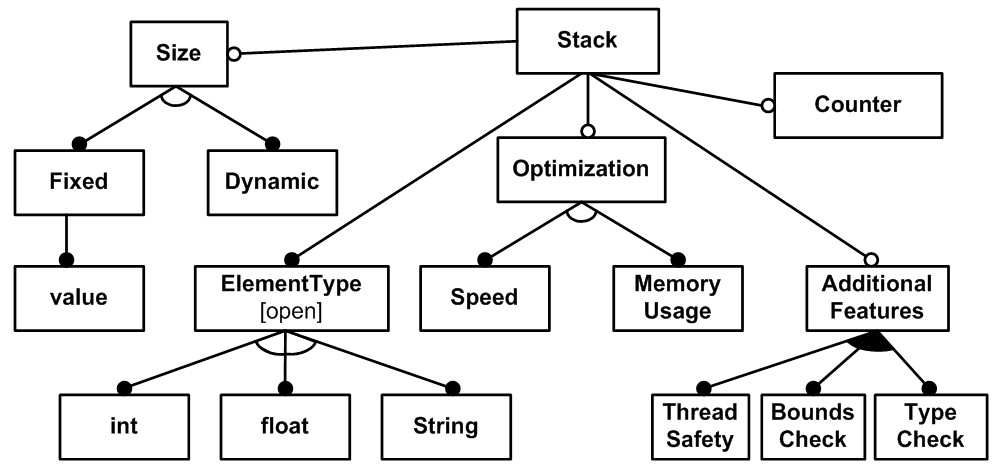
## DSLs and Product Lines

*This chapter discusses the role of DSLs in Product Line Engineering (PLE). We first briefly introduce PLE and feature models and discuss how feature models can be connected to programs expressed in DSLs. We then explain the difference in expressivity between feature models and DSLs and argue why sometimes features models are not enough to express the variability in a product line, and how DSLs can help. The chapter concludes with a mapping of the concepts relevant to PLE and DSLs. This chapter is written mostly for people with a background in PLE who want to understand how DSLs fit into PLE.*

### 21.1 Introduction

The goal of product line engineering (PLE) is to efficiently manage a range of products by factoring out commonalities such that definitions of products can be reduced to a specification of their variable aspects[[1]](#footnote-1). As a consequence of this approach, software quality can be improved and time-to-market of any single product in the product line can be reduced[[2]](#footnote-2). One way

of achieving this is the expression of product configurations on a higher level of abstraction than the actual implementation[[3]](#footnote-3). An automated mapping transforms the configuration to the implementation.

Figure 21.1: An example feature diagram for a product line of **Stack** data structures. Filled circles represent mandatory features, empty circles represent optional features. Filled arcs represent n-of-m selection and empty arcs represent 1-of-m.

|  |  |
| --- | --- |
| *21.2 Feature Models*  In PLE, this higher level of abstraction is typically realized |  |
| with feature models4. Feature models express configuration |  |
| options and the constraints among them. A graphical notation, called feature diagrams, is often used to represent feature models (Fig. 21.1 shows an example). Here is why we need constraints: if a product line’s variability was just expressed by a set of Boolean options, the configuration space would grow by 2*n*, with *n* representing the number of options. With feature models, constraints are expressed regarding the combinations of features, limiting the set of valid configurations to a more |  |
| manageable size. Constraints include5: |  |

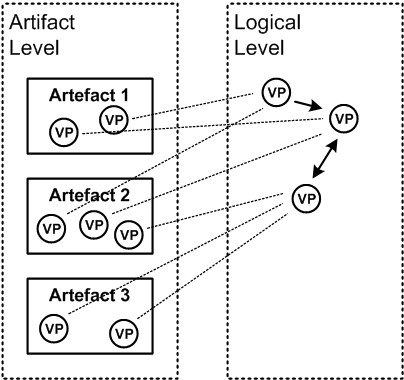
* *Mandatory* (filled circles): mandatory features have to be in each product. For example, in Fig. 21.1, each **Stack** has to have the feature **ElementType**.
* *Optional* (empty circles): optional features may or may not be in a product. **Counter** and **Optimization** are examples of optional features.
* *Or* (filled arc): a product may include zero, one or any number of the features in an **or** group. In the example, a product may include any number of features from **ThreadSafety**, **BoundsCheck** and **TypeCheck**.
* *Xor* (empty arc): a product must include exactly one of the features grouped into a **xor** group. The **ElementType** must either be **int**, **float** or **String**.

A *configuration* represents a product in the product line. It comprises a set of feature selections from a feature model that comply with the constraints expressed in the feature model.

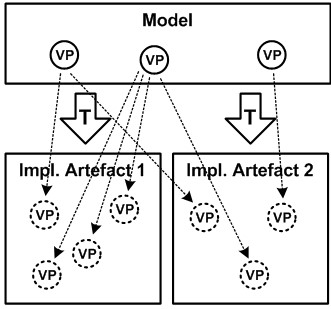
For example, the configuration **{Optimization, Memory Use,**

**Additional Features, Type Check, Element Type, int,**

|  |  |
| --- | --- |
| **Size, Dynamic}** would be valid. A configuration that includes **Speed** and **Memory Usage** would be invalid, because it violates the **xor** constraint between those two features expressed in the feature model.  Note that a feature model does not yet describe the *implementation* of a product or the product line, the feature model has to be connected to implementation artifacts in a separate |  |
| step6. |  |
| *21.3 Connecting Feature Models to Artifacts*  By definition, feature models express product line variability at a level that is more abstract than the implementation. In many systems, the implementation of variability is scattered over (parts of) many implementation artifacts. However, to result in a correct system, several variation points (VP) may need to be configured in a consistent, mutually dependent way. If each VP has to be configured separately, the overall complexity grows quickly. By identifying logical variation points and factoring them into features in a feature model, and then tying the (potentially many) implementation variation points to these logical variation points, related implementation variations can be tied together and managed as one (Fig. 21.2). |  |

).

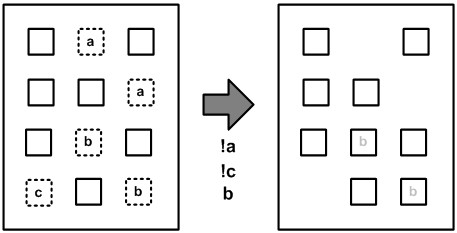
If DSLs are used to implement a software system, then the artifacts configured from the feature model are typically DSL programs, and the variation points are program elements. By using DSL models instead of low-level implementation code, the number of variation points in the artifacts will be reduced, because you use the DSL-to-code transformation to expand all the details in a consistent way. The trade-off is that you have to define this high-level domain specific language, including a way to define variants of programs written in that language. You also need to define the transformation down to the actual implementation artifacts (Fig. 21.3).



.

The configuration of models (and other artifacts) can be done in several different ways: removal, injection and parameterization.

|  |  |
| --- | --- |
| remove the part. These expressions are called *presence conditions*. The biggest advantage of this approach is its apparent simplicity. However, the comprehensive whole has to contain the parts for *all* variants (maybe even parts for combinations of variants), making it potentially large and complex. Also, depending on the tool support, the comprehensive whole might not even be a valid instance of the underlying language or for- |  |
| malism7. In an IDE, the respective artifact might show errors, |  |

 *Removal* (also known as *negative variability*) In this approach, the mapping from a feature model to implementation artifacts removes parts of a comprehensive whole (Fig. 21.4). This implies marking up the various optional parts of the comprehensive whole with Boolean expressions that determine when to

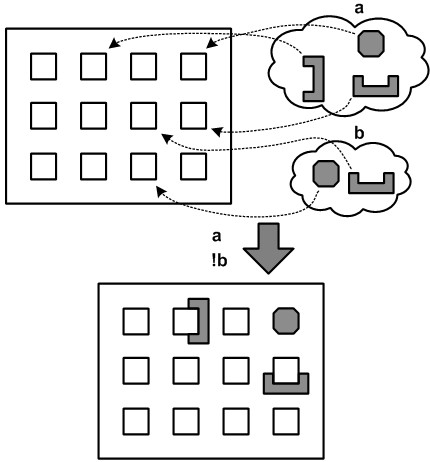
which makes this approach annoying at times.

**ifdefs** in C and C++ are a well-known implementation of this strategy. A preprocessor removes all code regions whose **ifdef** condition evaluates to false. When calling the compiler/preprocessor, you have to provide a number of symbols

that are evaluated as part of the conditions. Conditional compilation can also be found in other languages. Preprocessors that treat the source code simply as text are available for many languages and are part of many PLE tool suites. The AUTOSAR standard, as well as other modeling formalisms, support the annotation of model elements with presence conditions. The model element (and all its children) are removed from the model if the condition evaluates to false. The same approach is available in mbeddr.

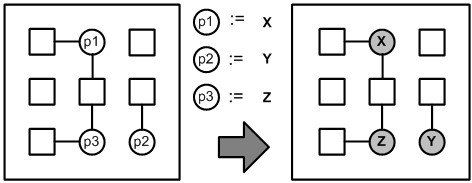
*Injection* (also known as *positive variability*) In this approach, additions are defined relative to a minimal core (Fig. 21.5). The core does not know about the variability: the additions point to the place where they need to be added. The clear advantage of this approach is that the core is typically small and contains only what is common for all products. The parts specific to a variant are kept external and added to the core only when necessary. To be able to do this, however, there must be a way to refer to the location in the minimal core at which to add a variable part. This either requires the explicit definition of named hooks in the minimal core, or some way of pointing into the core from an external source. Also, interactions between the additions for various features may also be hard to manage.

Aspect-oriented programming is a way of implementing this strategy. Pointcuts are a way of selecting from a set of join points in the base asset. A joint point is an addressable location in the core. Instead of explicitly defining hooks, all instances of a specific language construct are automatically addressable. Various preprocessors can also be used in this way. However,



they typically require the explicit markup of hooks in the minimal core. For models, injection is especially simple, since in most formalisms model elements are addressable by default and/or the language can be extended to be able to mark up hooks. This makes it possible to point to a model element, and add additional model elements to it, as long as the result is still a valid instance of the meta model.

*Parameterization* The variable artifact defines parameters. A variant is constructed by providing values for those parameters (Fig. 21.6). The parameters are usually typed to restrict the range of valid values. In most cases, the values for the parameters are relatively simple, such as strings, integers, Booleans or regular expressions. However, in principle, they can be arbi-



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | trarily complex8. The parameterized artifact needs to explicitly |  | | define the parameters, as well as a way to specify values. The artifact has to query the values of those parameters explicitly |  | | and use them for whatever it does. The approach requires the core to be explicitly aware of the variability9.  A configuration file that is read out by the an application is | . | | a form of parameterization. The names of the parameters are predefined by the application, and when defining a variant, a set of values is supplied. The Strategy pattern is a form of parameterization, especially in combination with a factory. A variant is created by supplying an implementation of an interface defined by the configurable application. All kinds of other small, simple or domain-specific languages can be used as a | . |  |  |  | | --- | --- | | the constraints defined in the feature model. Fig. 21.8 shows two example configurations. If an invalid configuration is created, errors will be shown in the configuration model. |  |   form of parameterization. A macro language in an application is a form of parameterization, where the type of parameter is "valid program written in language X"10.  *21.3.1 Example Implementation in mbeddr*  In mbeddr we use a textual notation for feature models. Fig. 21.7 shows this notation for the stack feature model shown graphically above. Note how the constraint affects all children of a feature, so we had to introduce the intermediate feature **options** to separate mandatory features from optional features. Features can also have configuration attributes (of any type).  A configuration is a named set of selections from the features in a feature model. The selection has to be valid regarding  . |

Presence conditions can be attached to any program element expressed in any language[[4]](#footnote-4), without this language having to

know about it, thanks to MPS’ annotations (discussed in Section 16.2.7). For example the two **report** statements and the message list in the left program in Fig. 21.9 are only part of a

product if the **logging** feature is selected in the product configuration. The background color of an annotated node is computed from the expression: annotated nodes using the same expression have the same color (an idea borrowed from Christian Kaestner’s CIDE12).

It is possible to edit the program as a product line (with the annotations), undecorated (without annotations), as well as a specific product. Fig. 21.9 shows an example. During transformation, those parts of programs that are not in the product are removed from the model.



2007

#### 21.3.2 Feature Models on Language Elements

Instead of using feature models to vary programs expressed with DSLs, the opposite approach is also possible. In this case, the primary product definition is done with DSLs. However, some language concepts have a feature model associated with them for detailed configuration. When the particular language concept is instantiated, a new ("empty") feature configuration is created, and can be configured by the application engineer.

#### 21.3.3 Variations in the Transformation or Execution

When working with DSLs, the execution of models – by transformation, code generation or interpretation – is under the control of the domain engineer. The transformations or the interpreter can also be varied based on a feature model.

*Negative Variability via Removal* The transformations or the interpreter can be annotated with presence conditions; the configuration happens before the transformations or the interpreter are executed.

*Branching* The interpreter or the transformations can query over a feature configuration and then branch accordingly at runtime.

*Positive Variability via Superimposition* Transformations or interpreters can be composed via superposition before execution. For transformations, this is especially feasible if the transformation language is declarative, which means that the order in which the transformations are specified is irrelevant. Interpreters are usually procedural, object-oriented or functional programs, so declarativeness is hard to achieve in those.

*Positive Variability via Aspects* If the transformation lan-

|  |  |
| --- | --- |
| guage or the interpreter implementation language support aspect oriented programming, then this can be used to configure the execution environment. For example, the Xpand code gen- |  |
| eration engine13 supports AOP for code generation templates.  Creating transformations with the help of other transformations, or by any of the above variability mechanisms, is also |  |
| referred to as *higher-order transformations*14. Note that if a boot- |  |
| strapped environment is used, the transformations are themselves models created with a transformation DSL. This case then reduces to just variation over models, as described in the previous subsection. |  |

### 21.4 From Feature Models to DSLs

A feature model is a compact representation of the features of the products in a product line, as well as the constraints imposed on combinations of these features in products. Feature models are an efficient formalism for *configuration*, i.e. for *selecting* a valid combination of features from the feature model. The set of products that can be defined by feature selection is fixed and finite: each valid combination of selected features constitutes a product. This means that all valid products have to be "designed into" the feature model, encoded in the features and the constraints among them. Some typical examples of things that can be modeled with feature models are the following:

* Does the communication system support encryption?
* Should the in-car entertainment system support MP3s?
* Should the system be optimized for performance or memory footprint?
* Should messages be queued? What is the queue size?

|  |  |
| --- | --- |
| necki16, one advantage of feature models is that a mapping |  |
| to logic exists. Using SAT solvers, it is possible to check, for example, whether a feature model has valid configurations at all. The technique can also be used to automatically complete partial configurations.  In the rest of this section we will discuss the limitations of feature models, in particular, that they are not suited for openended construction of product variants. Instead of giving up on models completely and using low-level programming, we should use DSLs instead. This avoids losing the differentiation between problem space and solution space, while still supporting more expressivity in the definition of a product.  As an example, we use a product line of water fountains, |  |
| as found in recreational parks17. Fountains can have several |  |

Because of the "select from set of options" metaphor, feature model-based configuration is simple to use – product definition is basically a decision tree. This makes product configuration efficient, and potentially accessible for stakeholders other than software developers. Also, as described by Batory15 and Czar-

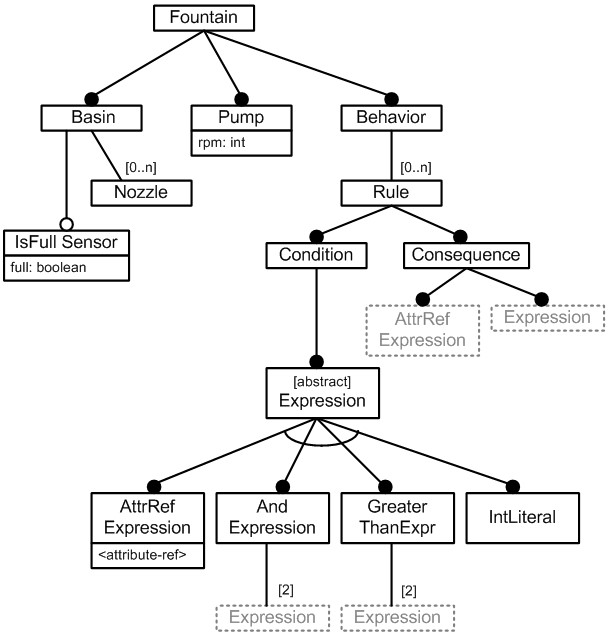
basins, pumps and nozzles. Software is used to program the behavior of the pumps and valves to make the sprinkling waters aesthetically pleasing. The feature model in Fig. 21.10 represents valid hardware combinations for a simple water fountain product line. Each feature corresponds to the presence of a hardware component in a particular fountain installation.

The real selling point of water fountains is their *behavior*. A fountain’s behavior determines how much water each pump should pump, at which time, with what power, or how a pump

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | reacts when a certain condition is met, e.g., a basin is full. Expressing the full range of such behaviors is not possible with feature models. Feature models can be used to select among a fixed number of predefined behaviors, but approximating all possible behaviors would lead to unwieldy feature models.  *21.4.1 Feature Models as Grammars*  To understand the limitations of feature models, we consider their relation to grammars. Feature models essentially corre- |  | | spond to context-free grammars without recursion18. For ex- |  | | ample, the feature model in Fig. 21.10 is equivalent to the following grammar19: |  |   Fountain -> Basin PUMP 19   |  |  | | --- | --- | |  |  | | This grammar represents a finite number of sentences: there are exactly four possible configurations, which correspond to the finite number of products in the product line. However, this formalism does not make sense for modeling behavior, for which there is typically an infinite range of variability. To accommodate for unbounded variability, the formalism needs to be extended. Allowing recursive grammar productions is sufficient to model unbounded configuration spaces, but for convenience, we consider also attributes and references.  *Attributes* express properties of features. For example, the  **PUMP** could have an integer attribute **rpm**, representing the power |  | | setting of the pump20. |  |   Fountain -> Basin PUMP(rpm:int) pure::variants.  Basin -> ISFULLSENSOR? (ONENOZZLE | TWONOZZLES)   |  |  | | --- | --- | | *Recursive* grammars can be used to model repetition21 and nest- | 21 Repetition is also supported by | |

ing. Nesting is necessary to model tree structures such as those occurring in expressions. The following grammar extends the fountain feature model with a **Behavior**, which consists of a number of **Rules**. The **Basin** can now have any number of

**Nozzles**.



Fountain -> Basin PUMP(rpm:int) Behavior

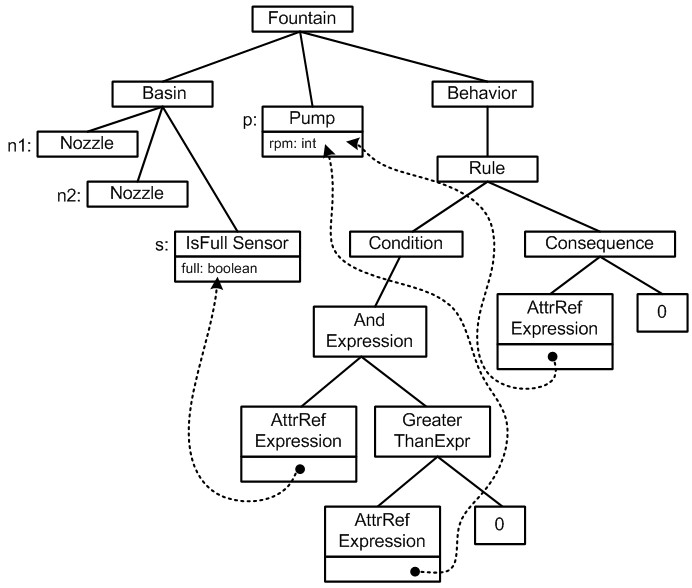
Basin -> ISFULLSENSOR? NOZZLE\* Behavior -> Rule\*

Rule -> CONDITION CONSEQUENCE

*References* allow the creation of context-sensitive relations between parts of programs described by the grammar. For example, by further extending our fountain grammar we can describe a rule whose condition refers to the **full** attribute of the **ISFULLSENSOR** and whose consequence sets a **PUMP**’s **rpm** to zero.

|  |
| --- |
| Fountain -> Basin id:PUMP(rpm:int)? Behavior  Basin -> id:ISFULLSENSOR(full:boolean)? id:NOZZLE\*  Behavior -> Rule\*  Rule -> Condition Consequence  Condition -> Expression  Expression -> ATTRREFEXPRESSION | AndExpression |  GreaterThanExpression | INTLITERAL;  AndExpression -> Expression Expression  GreaterThanExpression -> Expression Expression  Consequence -> ATTRREFEXPRESSION Expression |

Fig. 21.11 shows a possible rendering of the grammar with an enhanced feature modeling notation. We use cardinalities, as well as references to existing features, the latter are shown as dotted boxes. A valid configuration could be the one shown in Fig. 21.12. It shows a fountain with one basin, two nozzles named **n1** and **n2**, one sensor **s** and a pump **p**. It contains a rule that expresses the condition that if the **full** attribute of **s** is set, and the **rpm** of pump **p** is greater than zero, then the **rpm** should be set to zero.



#### 21.4.2 Domain-Specific Languages

While the extended grammar formalism discussed above enables us to cover the full range of behavior variability, the use of a graphical tree notation to instantiate these grammars is not practical. Another interpretation of these grammars is as the definition of a DSL – the tree in Fig. 21.12 looks like an abstract tree (AST). To make the language readable we need to add concrete syntax definitions (keywords), as in the following extension of the fountain grammar:

|  |
| --- |
| Fountain -> "fountain" Basin Pump Behavior  Basin -> "basin" IsFullSensor Nozzle\* Behavior -> Rule\*  Rule -> "if" Condition "then" Consequence  Condition -> Expression  Expression -> AttrRefExpression | AndExpression |  GreaterThanExpression | IntLiteral;  AndExpression -> Expression "&&" Expression  GreaterThanExpression -> Expression ">" Expression  AttrRefExpression -> <attribute-ref-by-name>  IntLiteral -> (0..9)\*  Consequence -> AttrRefExpression "=" Expression  IsFullSensor -> "sensor" ID (full:boolean)?  Nozzle -> "nozzle" ID  Pump -> "pump" ID (rpm:int)? |

We can now write a program that uses a convenient textual notation, which is especially useful for the expressions in the rules. We have created a DSL for configuring the composition *and* behavior of fountains22.

|  |
| --- |
| **fountain**  **basin sensor** s **nozzle** n1 **nozzle** n2  **pump** p  **if** s.full && p.rpm > 0 **then** p.rpm = 0 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | DSLs fill the gap between feature models and programming languages. They can be more expressive than feature models, but they are not as unrestricted and low-level as programming languages. Like programming languages, DSLs support *construction*, allowing the composition of an unlimited number of programs. Construction happens by instantiating language concepts, establishing relationships, and defining values for attributes. We do not a-priori know all possible valid pro- |  | | grams23. In contrast to programming languages, DSLs keep |  | | the distinction between problem space and solution space intact, since they consist of concepts and notations relevant to the problem domain. Non-programmers can continue to contribute directly to the product development process, without being exposed to implementation details.  *21.4.3 Making Feature Models More Expressive*  We described the limitations of the feature modeling approach above, and proposed DSLs as an alternative. However, the feature modeling community is working on alleviating some of these limitations. |  | | For example, cardinality based feature models24 support the |  | | multiple instantiation of feature subtrees. References between features could be established by using feature attributes typed with another feature – the value range would be the set of instances of this feature. Name references are an approximation of this approach. |  | | Clafer25 combines meta modeling and feature modeling. In | 25 K. Bak, K. Czarnecki, and A. Wa- | |

22 As we have discussed at length in this book, a complete language definition would also include typing rules and other constraints. However, to understand the difference in expressibility between DSLs and feature models, a

addition to providing a unified syntax and a semantics based on sets, Clafer also provides a mapping to SAT solvers to support validation of models. The following is an example Clafer[[5]](#footnote-5):

|  |
| --- |
| **abstract** Person name : String firstname : String **or** Gender Male  Female |

sowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *3rd International Conference on*

*Software Language Engineering*, 10/2010

2010

|  |
| --- |
| **xor** MaritalStatus  Single  Married  Divorced  Address  Street : String  City : String  Country : String  PostalCode : String State : String ?  **abstract** WaitingLine participants -> Person \* |

The code describes a concept **Person** with the following characteristics:

* A name and a first name of type **String** (similar to attributes).
* A gender, which is **Male** or **Female**, or both27 (similar to

**or**-groups in feature models).

* A marital status, which is either **single**, **married** or **divorced** (similar to **xor**-groups in feature models).
* An **Address** (similar composition is language definitions) .
* An optional **State** attribute on the address (similar to optional features in feature modeling).

The code also shows a reference: a **WaitingLine** refers to any number of **Persons**.

Note however that an important ingredient for making DSLs work in practice is the domain-specific concrete syntax. None of the approaches mentioned in this section provide customizable syntax. However, approaches like Clafer are a very interesting backend for DSLs, to support analysis, validation and automatic creation of valid programs from partial configurations.

### 21.5 Conceptual Mapping from PLE to DSLs

This section looks at the bigger picture of the relationship between PLE and DSLs. It contains a systematic mapping from the core concepts of PLE to the technical space of DSLs. First we briefly recap the core PLE concepts.

*Core Assets* designate reusable artifacts that are used in more than one product. As a consequence of their strategic relevance, they are usually high quality and maintained over time. Some of the core assets might have variation points.

*A Variation Point* is a well-defined location in a core asset where products differ from one another.

*Kind of Variability* classifies the degrees of freedom one has when binding the variation point. This ranges from setting a simple Boolean flag, through specifying a database URL or a DSL program, to a Java class hooked into a platform framework.

*Binding Time* denotes the point in time when the decision is made as to which alternative should be used for a variation point. Typical binding times include source time (changes to the source code are required), load time (bound when the system starts up) and runtime (the decision is made while the program is running).

|  |  |
| --- | --- |
| *The Solution Space* refers to the technical space that is used to implement the products. In the case of *software* product line |  |
| engineering28, this space is software development. The plat- |  |
| form lives in the solution space. The production tools create or adapt artifacts in the solution space based on a specification of a product in the problem space. |  |

*The Platform* is those core assets that actually form a part of the running system. Examples include libraries, frameworks or middleware.

*Production Tools* are core assets that are not part of the platform, but which are used during the (possibly) automated development of products.

*Domain Engineering* refers to activities in which the core assets are created. An important part of domain engineering is domain analysis, during which a fundamental understanding of the domain, its commonalities and variability is established.

*Application Engineering* is the phase in which the domain engineering artifacts are used to create products. Unless variation points use runtime binding, they are bound during this phase.

*The Problem Space* refers to the application domain in which the product line resides. The concepts found in the problem space are typically meaningful to non-programmers as well.

In the following sections we now elaborate on how these concepts are realized when DSLs are used.

#### 21.5.1 Variation Points and Kinds of Variability

This represents the core of the chapter and has been discussed extensively above: DSLs provide more expressivity than feature models, while not being completely unrestricted as programming languages.

#### 21.5.2 Domain Engineering and Application Engineering

As we develop an understanding of the domain, we classify the variability. If the variability at a particular variation point is suitable for DSLs (i.e. it cannot be expressed sensibly by pure configuration), we develop the actual languages together with the IDEs during domain engineering. The abstract syntax of the DSL constitutes a formal model of the variability found at the particular variation point29. The combination of several

DSLs is often necessary. Different variation points may have different DSLs that must be used together to describe a complete product30.

Application engineering involves using the DSLs to bind the respective variation points. The language definition, the constraints and the IDE guide the user along the degrees of freedom supported by the DSL.

#### 21.5.3 Problem Space and Solution Space

DSLs can represent any domain. They can be technical, in-

spired by a library, framework or middleware, expected to be used by programmers and architects. DSLs can also cover application domains, inspired by the application logic for which the application is built. In this case they are expected to be used by application domain experts. In the case of application DSLs, the DSL resides in the problem space. For execution they are mapped to the solution space by the production tools. Technical DSLs can, however, also be part of the solution space. In this case, DSL programs may be *created* by the mapping of an application domain DSL to the solution space. It is also possible for technical DSLs to be used by developers as an annotation for the application domain DSLs, controlling the mapping to the solution space, or configuring some technical aspect of the solution directly.

#### 21.5.4 Binding Time

DSL programs can either be transformed to executable code or interpreted. This maps to the binding times introduced above in the following way:

* If we generate source code that has to be compiled, packaged and deployed, the binding time is source. We speak of static variability, or static binding.
* If the DSL programs are interpreted, and the DSL programs can be changed as the system runs, this constitutes runtime binding, and we speak of dynamic variability.
* If we transform the DSL program into another formalism that is then interpreted by the running system, we are in the middle ground. Whether the variability is load-time or runtime depends on the details of how and when the result of the transformation is (re-)loaded into the running system.

#### 21.5.5 Core Assets, Platform and the Production Tools

DSLs constitute core assets; they are used for many, and often all, of the products in the product line. It is however not easy to answer the question of whether they are part of the platform or the production tools:

* If the DSL programs are transformed, the transformation code is a production tool; it is used in the production of the products. The DSL or the models are not part of the running system.
* In the case of interpretation, the interpreter is part of the platform. Since it directly works with the DSL program, the language definition becomes a part of the platform as well.
* If we can change the DSL programs as the system runs, even the IDE for the DSL is part of the platform.
* If the DSL programs are transformed into another formalism that is in turn interpreted by the platform, then the transformations constitute production tools, and the interpreter of the target formalism is a part of the platform.

1. PLE also involves a lot product management, process and organizational aspects. We do not cover these in this book. [↑](#footnote-ref-1)
2. It can also help establish a common user experience among a range of products. [↑](#footnote-ref-2)
3. This higher level of abstraction is often called the problem space; the lower level is often called the solution space or implementation space. [↑](#footnote-ref-3)
4. Note that the fact that you can make arbitrarily detailed program elements [↑](#footnote-ref-4)
5. adapted from Michal Antkiewicz’ *Concept Modeling Using Clafer* tutorial at **gsd.uwaterloo.ca/node/310** [↑](#footnote-ref-5)