University of Magdeburg
School of Computer Science

Bachelor Thesis

Improving Usability of UML Modeling Tools for Feature-Based Product Line Development

Author:

Maria Papendieck

January 10, 2011

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Janet Feigenspan

Department of Technical and Business Information Systems

University of Magdeburg
School of Computer Science
P.O. Box 4120, D–39016 Magdeburg, Germany

Papendieck, Maria:
Improving Usability of UML Modeling Tools for Feature-Based Product Line Development
Bachelor Thesis, University of Magdeburg, 2010.

Abstract

Software product line engineering (SPLE) and model-driven engineering (MDE) are powerful approaches that increase efficiency of the software engineering process. They can be combined to model-driven product line engineering (MDPLE). To further establish this combined approach, it is necessary to provide good tool support. To this end, we aim to improve the usability of an already existing tool for MDPLE: *pure::variants*, which is one of the leading tools for SPLE, with its connectors to the powerful UML tools *Rational Rhapsody (Rhapsody)* and *Enterprise Architect (Architect)*. Inspired by user requests, we identify usability problems of both Rhapsody and Architect when used with the connection to pure::variants. To solve the problems, we propose concepts for a UML tool extension (e.g., visualizations that improve UML model comprehension, and an editor that provides autocompletion and syntax highlighting for editing SPLE-related information of the UML model). To test whether the concepts can be realized, we implemented them for Rhapsody and Architect. During implementation, we encountered tool restrictions, such as the inability to apply visualizations to some parts of the UML tools, or to integrate our extension into Rhapsody's user interface. Nevertheless, we conclude, based on an evaluation of usability heuristics, that the implemented extensions improve Rhapsody's and Architect's usability for SPLE with pure::variants. Thus, our thesis contributes to the acceptance of MDPLE.

Contents

List of Figures

List of Tables

List of Abbreviations

Architect Enterprise Architect

CIDE Colored Integrated Development Environment

HCI Human-Computer Interaction

IDE Integrated Development Environment

ISO International Organization for Standardization

MDE Model-Driven Engineering

MDPLE Model-Driven Product Line Engineering

OMG Object Management Group

pvSCL pure::variants Simple Constraint Language

Rhapsody IBM Rational Rhapsody

SPL Software Product Line

SPLE Software Product Line Engineering

UI User Interface

UML Unified Modeling Language

1. Introduction

As the demand for complex and flexible software grows, software stakeholders consider it more and more important to create software products faster, while at the same time increasing their quality (Clements and Northrop [CN01], p. 17). Both *Software Product Line Engineering (SPLE)* and *Model-Driven Engineering (MDE)* have the potential to achieve these goals (van der Linden et al. [LSR07], p.287; Hailpern et al. [HT06]).

When using SPLE concepts, a higher efficiency of the software development process is reached by exploiting commonalities between multiple similar, yet different products. Instead of developing each product individually from scratch, they are engineered as one (software) product line, from which final products can be derived.

When employing MDE, on the other hand, the whole software engineering process is based on *models*, which can be described as "[...] representation[s] in a certain medium of something in the same or another medium" (Rumbaugh et al. [RJB04], p.13). Models can be used, for example, to get an overview of a project, to communicate its functions to customers, or to generate source code (Brown et al. [BCT05]). To specify models, developers can use a modeling language, such as the *Unified Modeling Language (UML)*, which is the current state of practice modeling language (Brown et al. [BCT05]).

Since both SPLE and MDE increase efficiency of the software engineering process in different ways, combining the approaches can be of advantage (Czarnecki et al. [CAK+05]). Thus, SPLE could facilitate the production of multiple products as one product line, which reduces time to market significantly, while MDE could help to represent different aspects of a product line more abstractly (Czarnecki et al. [CAK+05]). This relatively new concept is called *Model-Driven Product Line Engineering (MDPLE)*.

Despite the potential of MDPLE to increase the productivity and quality of software engineering processes significantly, its concepts are used reluctantly in industry [MBM09].

One reason for this can be lack of tool support, since well developed tool support is a necessary factor for the success of a software engineering approach (Steimann [Ste06]). Hence, it is necessary to implement new tools or improve existing tools to further establish the MDPLE approach.

There are only few tools already supporting MDPLE. For example, pure::variants[1] (pure-systems [pur04]) and Gears[2] (Krueger and Bakal [KB09]), the two leading tools for SPLE (van der Linden et al. [LSR07], p.310). Both tools provide connection to *IBM Rational Rhapsody (Rhapsody)*, which is a well developed UML modeling tool that can be used for MDE (Eichelberger et al. [EES09]). Apart from these commercial tools, there are MDPLE tools implemented in a research context. However, research projects typically do not provide a stable production environment, which is an important criterion for large software projects (van der Linden et al. [LSR07], p.310). Thus, we concentrate on pure::variants and Gears.

pure::variants and Gears both provide extensive support for developing Software Product Lines (SPLs), which can be combined with the large set of functions for MDE provided by Rhapsody. Because both the SPLE tools and the MDE tool are already very well developed, we suggest to improve the connection between the existing tools, rather than implementing a new tool.

Both connections to Rhapsody work basically the same: Users model one UML project with Rhapsody. With pure::variants or Gears they can derive other UML projects from the created project – the products of the SPL. To make the derivation of projects possible, users have to add SPL-related information to the model.

Since we have access to pure::variants, we analyzed its connection to Rhapsody for problems that would restrain users from working with the tool. Inspired by feature requests of pure::variants users, we identified several problems concerning the usability of Rhapsody: Rhapsody lacks support of actions related to SPLE with pure::variants. Therefore, the current way of adding or editing SPL-related information to a UML project is error-prone and inefficient.

In order to increase the acceptance of MDPLE, we suggest to improve the quality of pure::variants' connection to Rhapsody by improving Rhapsody's usability. Thus, we would improve tool support, which is a necessary factor for the success of a software engineering approach. Although Gears' connector to Rhapsody would also be a candidate for improving tool support, we choose only to improve the quality of pure::variants' connector, because we do not have access to Gears.

Additional to its connector to Rhapsody, pure::variants offers an equivalent connection to *Enterprise Architect (Architect)*, which is another well developed UML tool. Since Architect also lacks support for actions that are related to SPLE with pure::variants, we propose to improve its usability as well.

Based on these statements, we define goals that we aim to achieve in this thesis.

[1] http://www.pure-systems.com/
[2] http://www-01.ibm.com/software/awdtools/rhapsody/

Goals

Our main intent is to improve Rhapsody's and Architect's usability when used for SPLE with pure::variants, in order to contribute to the success of MDPLE. To improve the usability, we first have to identify usability problems in more detail, so that we can fix them. Therefore, we define our first goal:

1. A list of usability problems of UML tools used for SPLE with pure::variants

Based on the identified problems, we can then devise concepts how to improve the usability. Since Rhapsody and Architect lack support for SPLE-related actions, we suggest that they should be extended, such that their usability is improved for SPLE-related actions. Thus, the next goal is as follows:

2. Concepts for a UML tool extension that improves the usability of the tool

Finally, the concepts should be implemented. Hence, we define the last goal:

3. An implementation of the proposed extension for Architect and Rhapsody that improves the tools' usability

With this implementation, we can test whether the proposed concepts are possible to implement, and whether they improve the usability of Architect and Rhapsody when used for SPLE with pure::variants. If this is the case, we conclude that our extensions improve the quality of tool support for MDPLE, and thus, contribute to its success.

Structure

We structure the remainder of our thesis as follows: Chapter 2 contains the necessary background knowledge to understand the following chapters, which includes UML, SPLE, and *Human-Computer Interaction (HCI)*. In Chapter 3, we propose concepts for UML tool extensions that improve the tool's usability when used for SPLE. We present our implementation of these concepts for Architect and Rhapsody in Chapter 4, and evaluate in Chapter 5 whether the implemented evaluations improve the usability. We discuss related work in Chapter 6, and summarize our thesis in Chapter 7. Finally, we argue how to continue the work presented in this thesis.

2. Background

In this chapter, we introduce all background knowledge necessary to understand the remaining part of this thesis. To illustrate the introduced concepts, we use a running example, which we introduce in Section 2.1.

Our goal is to improve the *usability* of *UML* tools that are used for *SPLE*. Hence, we first introduce UML in Section 2.2, and SPLE in Section 2.3. Having explained these concepts, we describe the bridge between SPLE and UML with the example of pure::variants, a tool for SPLE. To explain usability, we first need to introduce HCI. We describe both concepts in Section 2.5.

2.1 The Weather Station Example

The Weather Station Example (Beuche and Dalgarno [BD06]) describe a home weather station used for simple measurements. It can measure *temperature*, *wind speed*, or *air pressure* of the weather station's environment. For each of the values, a separate *sensor* exists, which measures the current value - either temperature, wind speed, or air pressure - in a certain interval. Furthermore, it can emit warnings: A *heat warning*, when the temperature sinks or rises beyond a specified value, or a *gale warning*, when the measured wind speed exceeds a specified threshold.

There are many different weather station products for sale, ranging from basic to complicated, from cheap to expensive. The described example would still be a weather station, if it could only measure temperature, but not wind speed or air pressure. As long as one sensor remains, it can still be called a weather station. Thus, there can be many variants of weather stations.

Throughout the remaining chapter, we extend the introduced Weather Station Example with explanations and diagrams to better understand the presented concepts. Moreover, we use it in the main part of this thesis to visualize our concepts and implementation of a UML tool extension. In the next section, we introduce UML.

2.2 Unified Modeling Language

Here, we give a short overview of UML. We concentrate on aspects relevant for our thesis. A detailed overview can be found in *The Unified Modeling Language Reference Manual* [RJB04].

2.2.1 What is UML?

A formal definition by the Object Management Group (OMG) states that the *Unified Modeling Language* (UML) "[...] is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms" [UML10]. In short, this means UML is a visual modeling language providing elements and methods to model software systems.

UML modelers usually employ one or more models and diagrams to visualize these model(s). *Models* represent "a semantically complete abstraction of a system" (Rumbaugh et al. [RJB04], p.342). A UML model is build of semantic elements describing the modeled system. We refer to these elements as *model elements* in the remaining thesis. Users can view them in one or multiple diagrams. A *diagram* is "a graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements)" (Rumbaugh, Jacobsen and Booch [RJB04], p.260). A diagram itself is not a semantic element, but it shows representations of model elements. We call these representations *diagram elements*. They hold only graphical information on how the model element's representation is drawn, while the model element itself holds the actual semantics.

In the next section, we provide two diagrams describing the structure and behavior of the weather station introduced in Section 2.1. We explain all terms necessary to understand the presented diagrams.

2.2.2 Basic UML Elements

We start by describing the weather station's structure[1]. Figure 2.1 illustrates a *class diagram*. It shows nodes and connectors, the meaning of which we explain in this section. We use numbered labels, contained in the diagram, to refer to diagram elements during our explanation.

Label (1) indicates the class *TemperatureSensor*. A *class* "[...] represents a concept within the system being modeled" (Rumbaugh et al. [RJB04], p.185). This means, *TemperatureSensor* represents a concept of the weather station (the *system*) for measuring temperature. Like *WindSensor* and *PressureSensor*, it is connected to class *Sensor* (2) by an arrow (3). This arrow represents a *generalization*, which means all three classes inherit from *Sensor* (2). Below its name, *Sensor* contains two other entries:

[1] UML project available at: http://www.pure-systems.com/downloads/flash/pv-ea-intro/flash.html

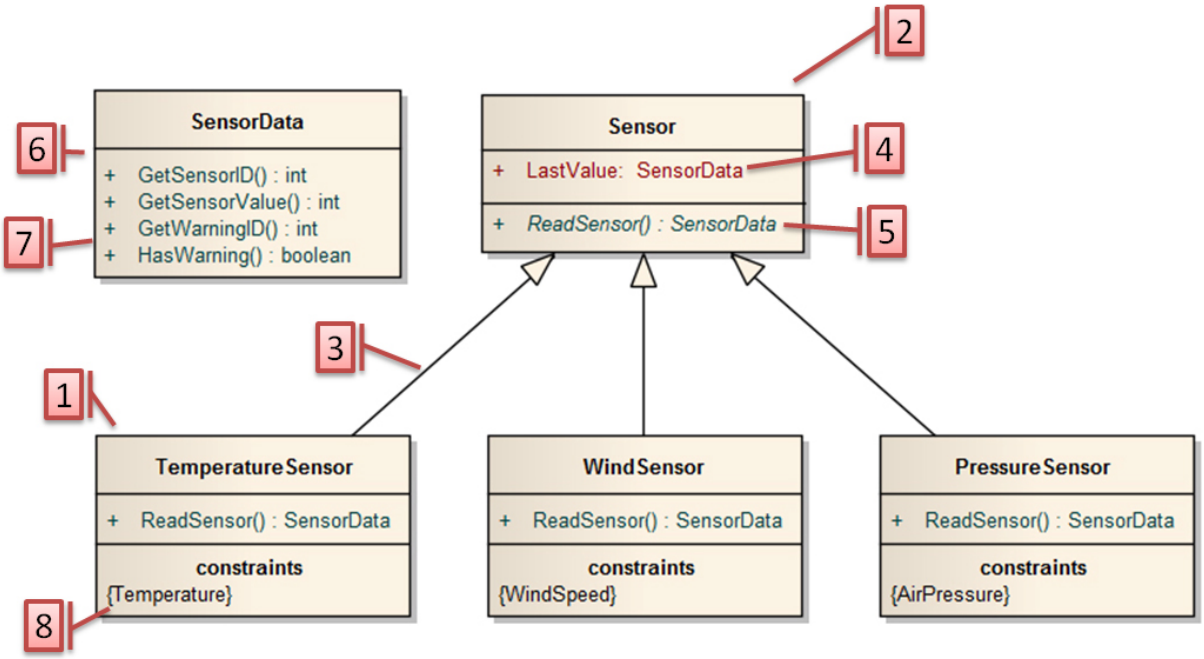


Figure 2.1: Class diagram of the Weather Station Example.

"+ LastValue: SensorData" (4) and "+ ReadSensor(): SensorData" (5). "+ LastValue: SensorData" (4) is an *attribute*, containing the last read value of the sensor, whereas "+ ReadSensor(): SensorData" (5) is an *operation*. It represents a method used to read the sensor's value. After the colon, the type of the attribute and operation are specified, respectively. The type "SensorData" refers to the class *SensorData* (6). It holds all information gathered during the reading process: the sensor's value and its type. Furthermore, it provides operations (7) to determine whether or not a warning occurred, and to get the warning's type.

Next, we have constraints, for example in class *TemperatureSensor* (8). *Constraints* are defined as "[...] boolean expression[s] represented as a string to be interpreted in a designated language" (Rumbaugh, Jacobsen and Booch [RJB04], p.58). This means they provide a way for a UML modeler to add custom information to any model element which can be used for making a decision. In our case, the constraint's text shows, which function a model element has: *TemperatureSensor* represents all temperature measuring functions, thus it is constrained by "Temperature". *SensorData's* operations "GetWarningID()" and "HasWarning()" are also constrained, even though the constraint cannot be displayed in the class diagram. Since they represent functions for warnings, they are constrained by "Warnings".

Having modeled the structure of the weather station in UML, it remains to describe its behavior. Thus, we illustrate a statechart in Figure 2.2. A *statechart* is a visual representation of a *state machine*, which "[...] models the possible life histories of an object of a class" (Rumbaugh, Jacobsen and Booch [RJB04], p.30). The state

machine of Figure 2.2 shows a sequence of all possible states the weather station can be in.

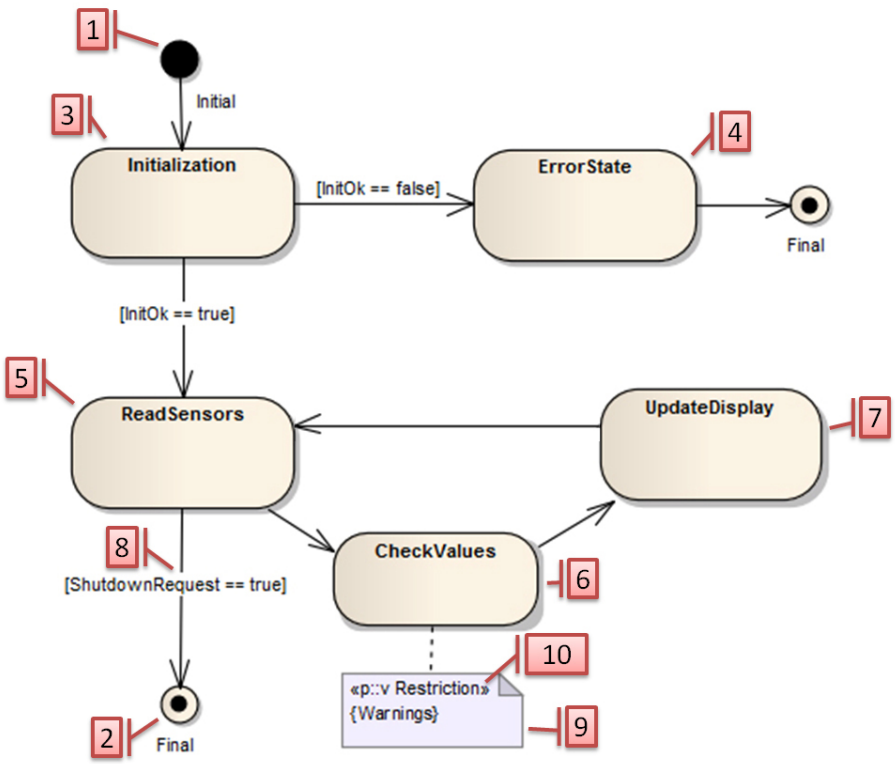


Figure 2.2: Statechart of the Weather Station Example.

It starts with its *initial state* (1) and ends when a *final state* (2) is reached. A *state* represents "a period of time during the life of an object" (Rumbaugh, Jacobsen and Booch [RJB04], p.30). For example, the states of the weather station are *Initialization*, *ErrorState*, *ReadSensors*, *CheckValues*, and *UpdateDisplay*. States are connected by *transitions*, which transfer the object to a new state if certain conditions are met. The conditions are usually specified by its label. In Figure 2.2, the weather station's life starts with the *Initialization* state (3). If it cannot be initialized due to an error, its current state changes to *ErrorState* (4) and its lifecycle ends. Otherwise, it reads the sensor's values (5), checks them for warnings (6), and displays the result (7), until the user wants to shut down the weather station (8). Similar to the class diagram, the statechart contains a constraint (9). It specifies, that *CheckValues* is related to the weather station's "Warnings" function. It also shows that constraints can have their own diagram element, other than in the class diagram of Figure 2.1.

There is an entry "≪p::v Restriction≫" placed above the constraints text (10). This indicates the constraint's *stereotype*, which is "[...] a kind of model element defined in the model itself" (Rumbaugh et al. [RJB04], p.103). Stereotypes are often used to extend UML for a particular domain. They always need a model element, which they can alter in different ways. In this case, we use the stereotype to distinguish the

constraint from other constraints without the "p::v Restriction" stereotype. Before we can explain the meaning of the "p::v Restriction" constraint, we need to introduce SPLE in the next section.

2.3 Software Product Line Engineering

Nowadays, software producers want to engineer software faster, at lower costs, and still with higher quality (Clements and Northrop [CN01], p. 17). A way to meet this demand is *Software Product Line Engineering* (SPLE). The idea is, instead of developing similar software products individually, to develop them as one SPL, exploiting their commonalities and differences. We explain the idea of SPLE in this chapter.

2.3.1 Software Product Lines

A Software Product Line (SPL) is most commonly defined as "[...] a set of software-intensive systems sharing a common, managed set of *features* that satisfy the specific needs of a *particular market segment or mission* and that are developed from a common set of *core assets* in a prescribed way" (Clements and Northrop [CN01], p. 5). To understand this definition, we explain the used terms.

The *domain*, which is described in the definition by "a particular market segment or mission", is an important term used when working with SPLs. It can be "[...] a specialized body of knowledge, an area of expertise, or a collection of related functionality" (Clements and Northrop [CN01], p.14). For example, the domain of the weather station we introduced in Section 2.1 is meteorology.

The definition also mentions features and assets, which are the basic components of an SPL. *Features* describe user-visible aspects of a product (Kang et al. [KCH+90], p.1), whereas *assets* refer to its software artifacts. These artifacts can be reused throughout the whole product line. The assets forming the base of an SPL are called *core assets* (Clements and Northrop [CN01], p.14), because they are used in every resulting product of the SPL. For the Weather Station Example, features are *temperature*, *wind speed*, *air pressure*, and *warnings*, since they represent its user-visible functions. Its assets, on the other hand, are the classes shown in Figure 2.1.

2.3.2 Feature Model

To represent all possible combinations of assets, feature models are usually used. They were first introduced by Kang et al., who state that "[...] a *feature model* represents the standard features of a family of systems in the domain and the relationships between them" ([KCH+90], p.36). To derive a final product from an SPL, a customer can use the feature model to select features she wants to have in her product. We call this combination of features a *configuration*.

To model dependencies of features, different types exist. We refer to these types as *variation types*. Features of type *and*, *or*, and *alternative* are organized in groups. Of

a group of features there have to be selected: *all* features, if the group's type is *and*, *exactly one* feature, if it is *alternative*, and *at least one*, if it is *or*. *Optional* features do not have to be selected, whereas *mandatory* features are always selected.

To demonstrate the types, we present a feature diagram of the Weather Station Example in Figure 2.3. A *feature diagram* is a graphical representation of a feature model, consisting of a hierarchy of features (Kang et al. [KCH+90], p.64). For convenience, we show the diagram in a simplified form. Features are represented by rectangles containing their name and their type below the name.

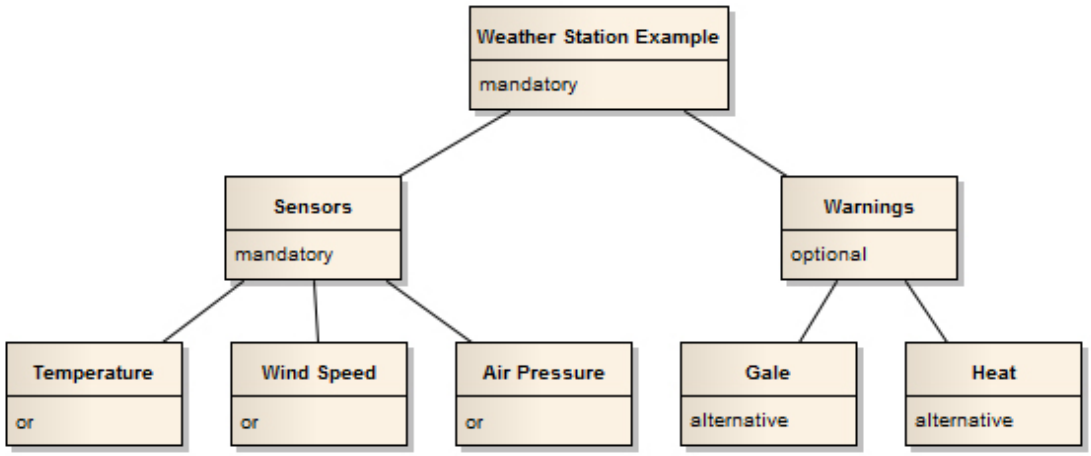


Figure 2.3: Simplified feature diagram of the Weather Station Example.

We first describe the feature *Sensors* and its children. Their types are *mandatory* and *or*, since the function of a weather station is to measure the value of at least one sensor. Using *mandatory* for *Sensors* ensures that *Sensors* is always selected, and using *or* for its child sensors ensures at least one type of sensor is included in the feature selection. This way, we assure that each configuration is a weather station, as described in Section 2.1. Since it is not necessary, but possible, for a weather station to emit warnings, the *Warnings* feature is optional. *Gale* and *Heat* are of type *alternative*, hence, if *Warnings* is selected, exactly one type of warning has to be selected.

Having explained the basic concepts of SPLE, we show which tools support developers of SPLs.

2.3.3 Tool Support

Developers of SPLs need to be supported in concurrently creating, maintaining, and using multiple versions of product line artifacts. Therefore, they require "[...] a development environment that facilitates the coordination of asset development and product development teams and processes and the sharing of assets among teams". (Clements and Northrop [CN01], p.208)

There exist only few tools for SPLE (van der Linden et al. [LSR07], p.309). They include *Gears* of BigLever Software[2] and *pure::variants* produced by pure-systems[3] in Magdeburg (Lutz [Lut07], van der Linden et al. [LSR07], p.309). Both, Gears and pure::variants support developers throughout the whole SPLE process and provide integration to standard tools (Lutz [Lut08]). Many other tools exist, which have mostly been developed from a research context. Hence, they typically do not provide a stable production environment (van der Linden et al. [LSR07], p.310). For further reading on SPLE tools, see [Lut07] or [Mat04].

Since we work with pure::variants in the implementation part of this thesis, we introduce it in the next section, and explain the basic steps of SPLE on its example.

2.4 pure::variants

pure::variants is a tool for feature-based (software) product line development. It is an extension to the successful Eclipse Integrated Development Environment (IDE) (Garvin [Gar09]). Clements and Northrop state that "fielding a product line involves *core asset development* and *product development* using the core assets, both under the aegis of technical and organizational *management*"([CN01],p.29). In this description, pure::variants fulfills the part of *management*. It supports users throughout the whole development and maintenance process of the product line (pure-systems [pur04]). To get a better understanding of what it is used for, we first describe the development process for a new product line and explain the used models. Finally, we describe pure::variants' UML extensions, since the implementation part of our thesis is based on their functions.

2.4.1 Developing Software Product Lines

There are three basic steps of the pure::variants development process for a new product line. First, the developers analyze problems and requirements of the SPL, and compile them into a *feature model*. Then, they specify the design of the solution, implement the core assets, and compile their structure into a *family model*. Last, they use a variant model to select features to be included in a desired product (pure-systems [pur04]). Next, we explain the models in more detail.

Feature Model

A pure::variants feature model is based on feature models as introduced in Section 2.3.2. Unlike the presented model, its features can only take four variation types: *or*, *alternative*, *optional*, and *mandatory*. In Figure 2.4, the feature model we outlined in Figure 2.3 is illustrated using the pure::variants tree layout. Features are shown as nodes of a collapsible tree. Each node has two icons. The first one indicates the variation type of the feature, which we summarize in Table 2.1. The second icon shows the type of the node. Since all of the nodes represent features, all have an icon showing an encircled "F".

[2] http://www.biglever.com/solution/product.html
[3] http://www.pure-systems.com/

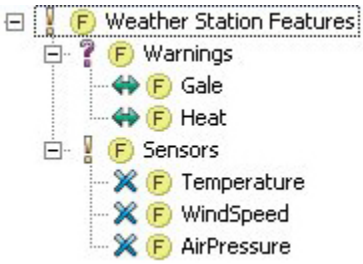


Figure 2.4: Feature model of the Weather Station Example displayed in pure::variants' tree layout.

Icon	Variability Type
	or
	alternative
	optional
	mandatory

Table 2.1: pure::variants variation types and according icons ([pur], p.103).

Family Model

While feature models describe problems and requirements of a product line, a *family model* manages its solution or implementation. It is used to "[...] describe the internal structure of the individual components of a product line and their dependencies on the features" (PURE-SYSTEMS [pur04]). For the remainder of our thesis, it suffices to know that family models store information concerning the implementation.

Variant Model

Finally, when a user wants to derive a product from a product line, she creates a configuration, which is specified by a *variant model* in pure::variants. Using this model, it is possible to select features that should be contained in the desired product. Here, the different feature types play an important role. They define which combinations of selected features are valid, and which are not. If a model is not valid, pure::variants marks the line of the variant model tree in which the error occurred.

In Figure 2.5, we display a variant model based on the feature model of Figure 2.4. Like the feature model, we present it using the pure::variants tree layout. The variant model is similar to the feature model, except for the checkboxes displayed left of each feature. They enable users to make a feature selection and support them in the selection process. For example, the ☒ left of *Heat* indicates that it is non-selectable, because *Gale* is already selected. Since *Gale* and *Heat* are alternative features, only one can be selected.

This concludes the explanation of pure::variants' SPLE process and its models. Next, we describe pure::variants' UML extensions.

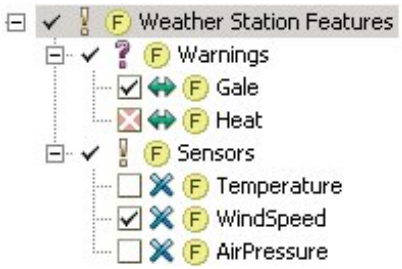


Figure 2.5: Variant model for the Weather Station Example.

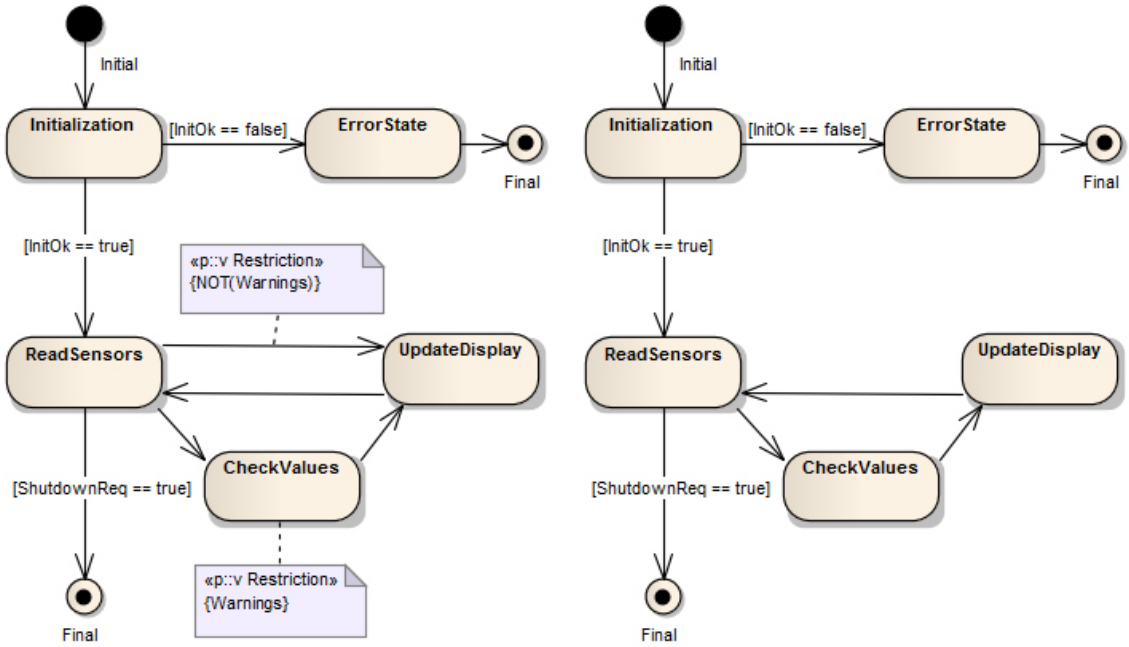
2.4.2 UML Extensions

pure-systems offers two extensions[4] for pure::variants, which make it possible to develop UML models as a product line. This functionality is available for the UML standard tools IBM *IBM Rational Rhapsody (Rhapsody)*[5] and Sparx Systems *Enterprise Architect (Architect)*[6]. Since the implementation part of our thesis is based on these extensions, we explain them in more detail. Both extensions have the same base functionality. They only differ in the UML tool they employ. Hence, we explain the base functionality of pure::variants' Architect and Rhapsody extension together.

The pure::variants UML extensions allow users to work on one UML *master project*, from which other projects can be derived (just like products from an SPL). The master project can either be created with Architect or Rhapsody. In order to support the process of deriving projects, it contains information on how its elements are related to features (*variability information*). Developers can add this information to a master project by using constraints with a "p::v Restriction" stereotype. For example, all constraints of Figure 2.1 and the constraints of Figure 2.6(a) contain variability information. When deriving a project, pure::variants processes these constraints to decide which model elements to delete in the derived project. In order to decide this, the contraint's text is given in the *pure::variants Simple Constraint Language (pvSCL)*, a simple language for expressing constraints and restrictions, providing logical and relational operators to build boolean expressions ([pur], p. 120). When pure::variants processes a pvSCL expression, it uses this expression to answer the question: "Should the constrained elements be contained in the derived project?". After this step, no variability information is contained in a project anymore.

For better understanding, we illustrate the presented concepts with the example shown in Figure 2.6(a) and Figure 2.6(b). Figure 2.6(a) displays the statechart of a master project, whereas Figure 2.6(b) depicts a statechart derived from (a) using the configuration of Figure 2.5. The statechart of the master project contains two pure::variants constraints: "Warnings" and "NOT(Warnings)". "Warnings" is a simple pvSCL expression containing only a feature name. "NOT(Warnings)" contains the pvSCL keyword

[4]http://www.pure-systems.com/index.php?id=21&L=0
[5]http://www-01.ibm.com/software/awdtools/rhapsody/
[6]http://www.sparxsystems.com.au/



(a) Statechart of Master Project (b) Derived Statechart

Figure 2.6: Statechart of the master project (a) and a derived statechart (b) using the variant model of Figure 2.5.

NOT with the *Warnings* feature as parameter. Thus, if *Warnings* is selected in a configuration, the constrained model elements are omitted in the resulting project. Since *Warnings* is selected in the configuration of Figure 2.5, the derived statechart of Figure 2.6(b) contains *CheckValues*, but does not contain the transition constrained with "NOT(Warnings)". Hence, the weather station represented by the derived statechart of Figure 2.6(b) always checks the read values before it updates its display. If the value is invalid, it updates the display with an error message. Furthermore, the statechart does not contain any constraints, because variability information is only contained for processing the master project.

In the next section, we provide necessary information on HCI with focus on usability.

2.5 Human-Computer Interaction

Human-Computer Interaction (HCI) is defined as "[...] a discipline concerned with the *design*, *evaluation* and *implementation* of interactive computing systems for human use and with the study of major phenomena surrounding them" (Hewett et al. [HBC+92], p.5). Since we aim to *design*, *implement*, and *evaluate* a UML tool extension, we explain HCI in more detail. We focus on aspects relevant for our thesis, which include: Usability, because we aim to improve the extension's usability; human characteristics, such as human vision, which we can utilize for improving usability; and

User Interface (UI) elements, which we use for building the extension's UI. We start with explaining usability.

2.5.1 Usability

Usability is "[...] a quality attribute relating to how easy something is to use" (Nielsen et al. [NL06], p.xvi). It is traditionally composed of different usability attributes that can be used to evaluate or measure the usability of a system.

2.5.1.1 Usability Attributes

There are many different definitions of usability and its attributes, for example by Shackel [Sha91], the International Organization for Standardization (ISO) [ISO98], Lindgaard [Lin94], or Nielsen [Nie94b]. We decide to use Nielsen's definition, since he subdivides usability into precise and measurable components, such that it can be systematically approached and evaluated (Nielsen [Nie94b]). He defines five usability attributes: Learnability, efficiency, memorability, error rate, and satisfaction (Nielsen [Nie94b], p.26). Throughout the remaining thesis, we refer to these usability attributes when reasoning about how to improve the usability of UML tools for SPLE, and when evaluating the usability of our implementation. Thus, we explain them in more detail in the following list:

1. *Learnability* relates to how easy something is to learn, such that users can start to work with a system quickly. Previous knowledge (e.g., when users are familiar with a former version of a system) can reduce the time needed for learning significantly (Nielsen [Nie94b], p.26,28).

2. *Efficiency* refers to how many cognitive resources are expended in relation to the accuracy and completeness with which users achieve goals [ISO98]. A system should be efficient to use, so that a high level of productivity is possible after the user has learned the system's functionality (Nielsen [Nie94b], p.26).

3. *Memorability* describes how easy it is to remember the functionality of a system. When users have not used a system for some time, they should be able to return to it without having to learn its functionality again (Nielsen [Nie94b], p.26).

4. The *error rate* of a system refers to how many errors users make while using the system. For good usability results, the error rate should be low. If errors occur, users should be able to recover from them easily. Catastrophic errors – such as a crash of the system, such that users loose all unsaved work, or small errors that are not discovered by users and lead to faulty results – should not occur (Nielsen [Nie94b], p.32-33).

5. *Satisfaction* relates to how much users like using a system. It is a subjective attribute, since different users can be differently pleased with a system (Nielsen [Nie94b], p.33). Subjective satisfaction can be especially important for systems used in a non-work environment, because users are working with the system for entertainment (Virzi [Vir91]).

To achieve these attributes, NIELSEN advises to follow usability principles, which we explain next.

2.5.1.2 Usability Heuristics

Optimizing the introduced usability attributes improves the overall usability of a system. To indicate usability problems and fix them, NIELSEN and MOLICH introduced the concept of usability heuristics [MN90][NM90]. These heuristics are "[...] a small set of fairly broad usability principles" (NIELSEN [Nie94a]). When followed, they help developers in creating usable UIs, and when used to analyze a system, they help identifying usability issues. For consistency, we use NIELSEN's refined heuristics introduced in [Nie94a]. Other heuristics can be found in [Shn97] by SHNEIDERMAN or [GP96] by GERHARDT-POWALS. In the following enumeration, we explain NIELSEN's heuristics.

1. *Visibility of system status*: A system should always inform users about what it is doing. Each user input should be followed by a reaction of the system. Especially, when a system has long response times, feedback is important. When the delay is greater than 10 seconds, users have to be informed of how long they have to wait, for example by a progress indicator (NIELSEN [Nie94b], p.135f; CARD ET AL. [CRM91]).

2. *Match between system and real world*: A system should employ the users' terminology to communicate with them, rather than using its internal terminology (NIELSEN [Nie94b], p. 123). Moreover, it should make use of users' background knowledge (NIELSEN [Nie94a]).

3. *User control and freedom*: Users should always feel in control of the system they are using. For example, they should always be able to undo or redo an action, close an open dialog, or interrupt a process that takes too long (NIELSEN [Nie94b], p.138,139).

4. *Consistency and standards*: An interface should be consistent, which means that required actions should be consistent in similar situations. Terminology in prompts, menus and help screens should be identical, and consistent color, layout, fonts etc. should be used (NIELSEN [Nie94a]; SHNEIDERMAN [Shn97], p.74).

5. *Error prevention*: UI elements should be chosen such that the error rate is kept at a minimum (NIELSEN [Nie94a]; SHNEIDERMAN [Shn97], p.75).

6. *Recognition rather than recall*: Users are much better at recognizing something than at remembering it without help. Hence, a system should rely on UI elements letting users choose from a list of possible items, or letting users modify data rather than entering it completely from scratch (NIELSEN [Nie94b], p.129). For example, typing errors can be reduced by using proposals, because users have to type less.

7. *Flexibility and efficiency of use*: A system should be flexible and efficient to use. This can be achieved, for example, by providing shortcuts to frequently used functionality (Nielsen [Nie94a]).

8. *Aesthetic and minimalistic design*: The design of a system should be aesthetic, so that users like using the system. Since the number of information pieces processed in short-term memory is limited to seven plus or minus two units (Miller [Mil56]) displays and the system's design should be kept as simple as possible. (Nielsen [Nie94a])

9. *Error recognition and recovery*: When an error occurs, the system should detect the error immediately and offer simple, constructive, and specific instructions for recovery (Nielsen [Nie94a];Shneiderman [Shn97], p.75).

This concludes the necessary information on usability. Next, we give an overview of of human perception.

2.5.2 Human Perception

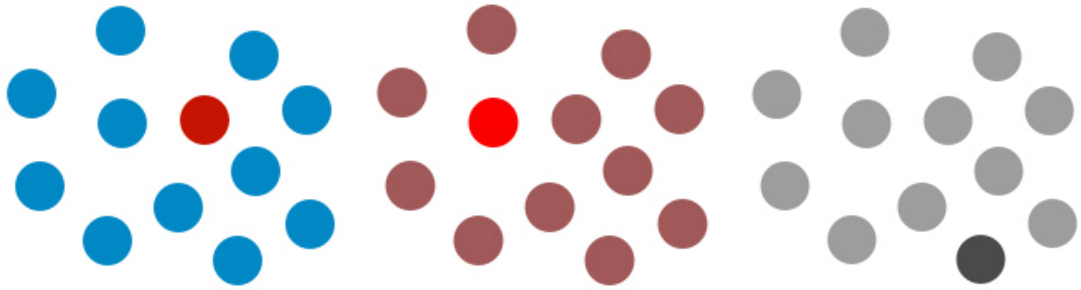
In our approach to improve the usability of UML tools used for SPLE, we primarily rely on visualizations, because they provide a very efficient way to discover characteristics or errors in data (Ware [War04], p.2). To produce good visualizations, it is necessary to know how they are processed by users. Hence, we give a short overview of the human perception process. We concentrate on aspects relevant for our thesis. A detailed overview can be found in *Information Visualization - Perception for Design* by Colin Ware [War04].

The process of human perception starts with light falling onto the retina of the human eye. The eye‘s retina is covered with two kinds of receptors: approximately six million *cones* for color vision and 120 million *rods* for black-and-white vision (Goldstein [Gol10], p.50). At the *fovea* – a point at the center of the retina – the density of cones is much higher than that of rods, whereas in the periphery it is much lower (Goldstein [Gol10], p.50). Thus, the point with the best color vision and resolution is located in the fovea. To perceive a complete image – rather than a small region – with best color vision and resolution, the eye has to be moved. This movement takes at least 200 ms (Diehl [Die07], p.18). Good visualizations are designed such that they save this time.

Treisman and Gelade describe the perception of an object in two stages [TG80]: The preattentive stage followed by the focused attention stage. During *preattentive stage*, no attention is needed, which means that visual features of an object can be perceived without moving the eye and thus, very fast. The object is broken down into *visual features*, such as its size, orientation, or location. The *focused attention stage*, on the other hand, depends on attention, which means that the eye has to be moved, and thus, 200 ms are necessary. During this stage, we perceive the whole object, not its individual features. (Goldstein [Gol10], p.144)

This knowledge can be utilized to produce good visualizations: Since visual features of the preattentive stage are perceived before moving the eye, these visual features provide the fastest way to communicate information to the viewer of a visualization. For example, marking the current location on a map with a red dot can help users in finding out where they are. The visual features in this example are the dot's *shape*, *size*, and *color*, or more precisely, its *hue*, since color consists of multiple visual features. We explain the meaning of hue next.

Color is composed of three preattentively perceived visual features: hue, saturation, and value. These terms represent a color in the HSV color space (Smith [Smi78]). In Figure 2.7, we demonstrate examples using these visual features to distinguish between different dots. In Figure 2.7(a), we use the visual feature *hue*, which represents an approximation of the visible spectrum ranging from red over yellow and green to blue and then back to red (Ware [War04], p.128). In the example, the red dot can be distinguished from blue dots without directing attention to it. Figure 2.7(b) shows dots that can be separated into two groups by their saturation. *Saturation* measures how "intensive" a hue is (Ware [War04], p.128). The pure red dot has the highest saturation value possible, whereas the rest of the dots have a low saturation. In Figure 2.7(c), the value of the dots is used to distinguish between them. *Value* refers to how dark or light a color is (Ware [War04], p.128). In Figure 2.7(c), we use it to highlight the dark dot and lowlight the rest. *Highlighting* means that the dot attracts viewers' attention – it is emphasized. *Lowlighting* is the opposite of highlighting, since it de-emphasizes elements. Highlighting and lowlighting of dots is possible in Figure 2.7(c), because humans perceive darker values as more important, or closer, than lighter values ([Ber83], p.73).



(a) hue (b) saturation (c) value

Figure 2.7: Demonstration of preattentive vision using hue (a), saturation (b) and value (c).

Based on visual features of preattentive stage, Bertin introduced eight *visual variables* that can be used to encode information in a visualization: Size, shape, orientation, horizontal/vertical position, hue, value, and texture ([Ber83], p.42). We use some of these variables for our visualizations in the next chapters. Apart from visualizations, users need to have access to the system's functionality. Therefore, we explain UI elements providing this access, next.

2.5.3 User Interface Elements

In this section, we explain common UI elements with the example of Enterprise Architect (Architect), which is one of the UML tools we extend. We omit a description of the other UML tool (IBM Rational Rhapsody (Rhapsody)), because its UI elements are equivalent to those of Architect.

Since the fourth heuristic introduced in Section 2.5.1.2 states that a UI should be consistent, the UI of our UML tool extension should be consistent with the UI of the extended tool. Furthermore, it is useful to rely on users' previously learned knowledge of UI functionality to improve the UI's learnability (Nielsen [Nie94b], p.26,28). In order to create a UI consistent with Architect's UI, we need to know its UI. To this end, we describe Architect's UI elements based on the screenshot displayed in Figure 2.8. It is labeled for better reference. We only give a short overview of common UI elements. A detailed description can be found in *The Essential Guide to User Interface Design* [Gal02].

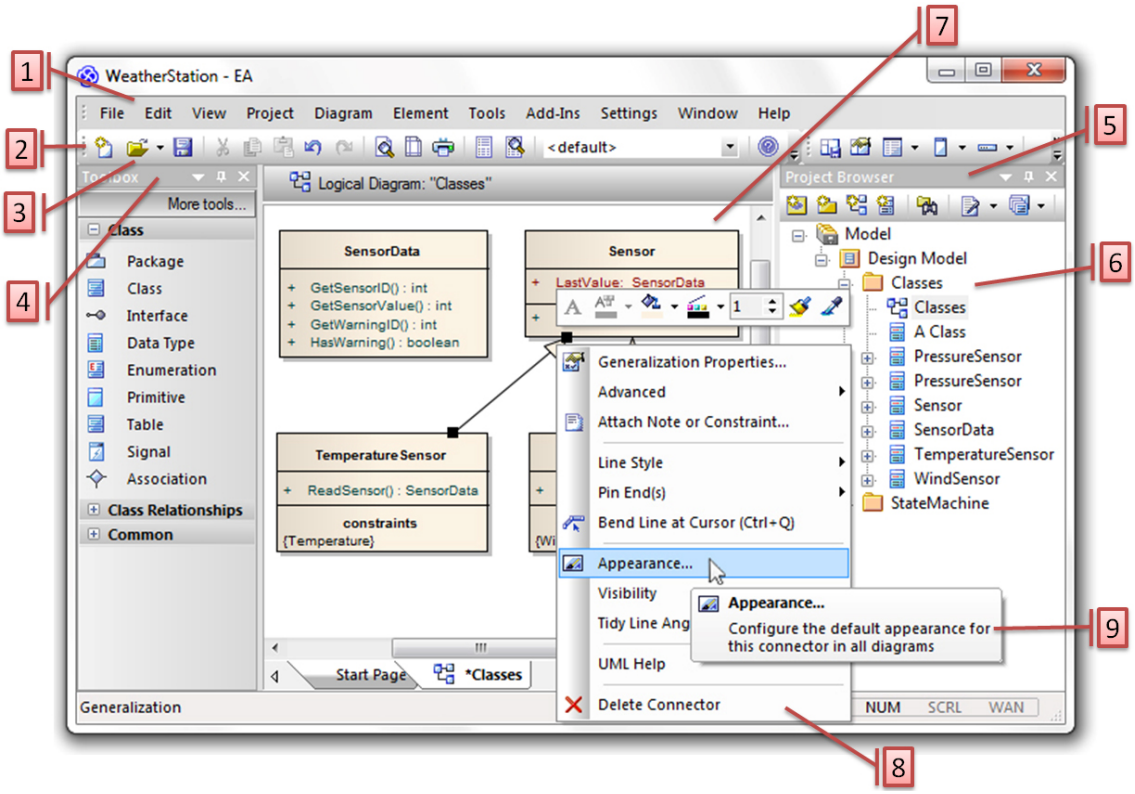


Figure 2.8: The user interface of Enterprise Architect.

Architect's UI is contained in a *window*. Label (1) indicates the window's *menubar*, with which users can navigate hierarchically through Architect's features. Below the menubar, *toolbars* are positioned (2). They contain *buttons* and other UI elements providing shortcuts to Architect's features. Users can click buttons to trigger an action or toggle them to switch between two states. A *toggle button* is marked to be selected

or unselected according to its state. Buttons can contain text and/or an icon, although in toolbars, they often only contain an *icon* – a small picture describing the triggered action in an easy-to-understand way. A special kind of button is the *split button* (3). Next to its icon, a small triangle is drawn. When a user clicks the triangle, a menu opens providing further selection possibilities related to the button's action.

In addition to toolbars (2), *toolpanes* (4) exist. Like toolbars, they can be dragged from one location and dropped to another. One of these toolpanes is the *Project Browser* (5). It contains a *model tree* (6), which displays the model elements of a UML project in a hierarchical representation. Another special feature of UML tools is the *diagram view* (7). It displays the diagrams of a UML project. Furthermore, users can right-click each diagram element to open a *context menu* (8). It is similar to the menus of the menubar, but can be opened by right-clicking on an object (in our case, the selected generalization). It only shows menu entries relevant to the selected object. In Figure 2.8, the mouse pointer is located above the menu entry "Appearance...". When hovering the mouse pointer over some UI elements, a *tooltip* opens (9), containing additional information about the hovered element.

This concludes all background information necessary to understand the remaining thesis. In the next chapter, we propose concepts for extending UML tools used for SPLE. We focus on improving the tools' usability based on the information provided in this chapter.

3. Concepts for a UML Tool Extension

Having introduced all necessary background knowledge, we address the first two goals we set in Chapter 1:

1. A list of usability problems of UML tools used for SPLE with pure::variants
2. Concepts for a UML tool extension that improves the usability of the tool

Usability issues arise when using Rhapsody or Architect for modeling a master project of a pure::variants product line. Hence, we devise concepts for an extension to a UML tool, such as Architect or Rhapsody, that solve these issues.

First, we give a short overview of the usability issues we identified. Then, we address these issues. In each section, we describe which usability issues we address, explain them in more detail, and propose concepts to solve them. In Section 3.2, we present concepts for visualizations supporting the modeling process of master projects. Then, we discuss how users can be supported in editing variability information of a UML project. Finally, we propose a UI allowing users to interact with our UML tool extension.

3.1 Usability Issues

In the following list, we give an overview of the usability issues we found in Architect and Rhapsody when used for SPLE with pure::variants. They are inspired by feature requests made by users of pure::variants' connection to Architect and Rhapsody. We identified the issues by testing the tools with focus on the usability heuristics introduced in Section 2.5.1.2. We explain the issues in more detail when we address them in the following sections.

1. Users have to switch between pure::variants and the UML tool to view a derived UML project, which conflicts with heuristic *flexibility and efficiency of use*.
2. Existing errors can only be found by scanning each pvSCL expression individually, which also contradicts heuristic *flexibility and efficiency of use*.
3. Users have to enter pvSCL expressions without support, which conflicts with heuristic *error prevention* and heuristic *recognition rather than recall*.
4. Errors in pvSCL expressions are not reported. This contradicts usability heuristic *error recognition and recovery*.

Next, we propose concepts to solve these issues. We start with suggesting visualizations to address the first and second issue.

3.2 Visualizations

The current pure::variants approach to develop UML projects as a product line does not support developers in modeling a master project, since pure::variants only processes completed master projects. This results in decreased usability of the UML tools when used for SPLE with pure::variants. We aim to improve their usability by extending the UML tools with visualizations that support understanding of the master project's connection to the product line. To achieve this, we plan to process the project's variability information to produce visualizations, which increase efficiency and reduce errors made during modeling the master project.

Before we present specific visualizations, we reason which visual variables to use, and describe how visualizations can be applied to UML projects.

3.2.1 Choice of Visual Variables

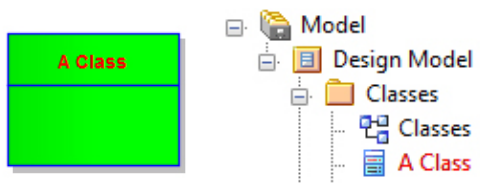
Since one goal of this thesis is to visualize information in UML tools, it is necessary to decide how to encode this information. A UML model consists of model elements, which are viewed and manipulated by users. Hence, we choose to apply our visualizations to the project's model elements. UML tools provide two ways of viewing model elements in UML projects: diagrams and model trees (cf. Section 2.5.3). Therefore, visualizations should be applicable to both diagrams and model trees.

There are eight elementary visual variables for encoding information graphically: *Shape*, *orientation*, *size*, *horizontal/vertical position*, *texture*, *hue* and *value* (cf. Section 2.5.2; [Ber83], p.42). For our visualizations, we rule out *shape* and *orientation*, because they would conflict with UML's graphical notation. *Size* or *position* would change the diagram layout, hence, we cannot use them for our visualizations. Moreover, *position* cannot be used in visualizations applied to the UML model tree, since its layout already uses *position* to encode the tree's hierarchy information. Also difficult to apply to model tree elements is *texture*. Thus, only *hue* and *value* remain.

Since the alternatives are generally worse, color is considered extremely effective to enable viewers to classify visual objects into separate categories (Ware [War04], p.133). Furthermore, *hue* and *value* can both be perceived preattentively, which results in a significantly reduced perception time for the visualized data (Ware [War04], p.165-166). For these reasons, we choose *hue* and *value* to encode visualization data. In the next section, we specify how to apply the chosen visual variables to model elements.

3.2.2 Formatting Model Elements

To use visualizations, we need a means to apply them. We choose to apply the visualizations to model elements of a UML model. Hence, we need to specify how to format the model elements to take hue and value used in our visualization. Model elements in UML tools, such as Architect and Rhapsody, are represented either in the model tree or in diagrams. In Figure 3.1, we illustrate how to color diagram elements (a) and elements of the model tree (b). Red represents the *foreground color*, green is the *background color*, and blue the *border color* of the element.



(a) (b)

Figure 3.1: Example for manipulating the colors of a diagram element (a) and a model element displayed in the model tree (b).

Apart from the foreground color of a model tree element, it has an icon, e.g., (cf. Section 2.5.3). This icon could also be colored, but we choose not to do so, because changing the color of an icon might change its meaning. On the other hand, to lowlight an element (cf. Section 2.5.2), all of its sub-elements including its icon should be lowlighted, so that it does not attract users' attention. The icon could be desaturated, and would still be recognizable by its form (e.g.,). Hence, we propose to change the color of an icon only when lowlighting it. Furthermore, the border and foreground color of a diagram element should always have a different value than the element's background, even though all colors have the same hue. Thus, viewers are still able to recognize its original layout, despite the changed colors.

Next, we present different types of visualizations we use in our UML extension.

3.2.3 Visualization Types

When modeling a master project for a product line of UML projects, it is necessary to define which model element is related to what feature(s), because this information is needed to derive products from the master project. This *element-feature relation* is

specified by each model element's variability information. When using pure::variants, the variability information consists of constraints containing pvSCL expressions. To support developers of a master project, an important task is to visualize the element-feature relation.

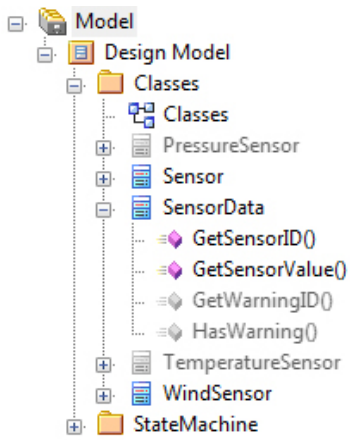
We present two different visualizations. We start with addressing the first usability issue specified in Section 3.1 (i.e., necessary switching between pure::variants and the UML tool to view a derived UML project). We present concepts for visualizing element-feature relations in the context of a *configuration* of features. Then, we propose how to visualize *errors* in the variability information of the master project to solve the second usability issue (i.e., detection of existing errors only possible through scanning pvSCL expressions individually).

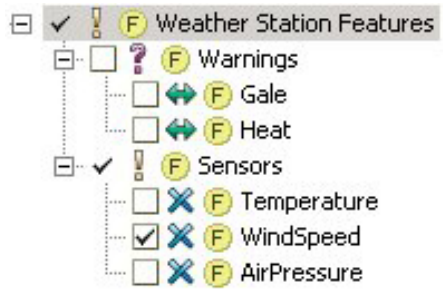
3.2.3.1 Visualizing Configurations

Deriving a UML project from a UML master project means deleting model elements from the master project, based on a configuration. When using pure::variants, currently the only way available to test whether the variability information of a master project is correct, is to derive a project and search for errors by examining the resulting project. The derivation of a project can only be done in pure::variants when the master project is completed. pure::variants loads the file of the master project; Therefore, the UML project has to be closed. This switching between tools costs time, and thus, reduces efficiency of the master project modeling process. To this end, we propose to extend the tool with a visualization of configurations, which enables users to preview the result of a configuration without changing the master project or having to close it.

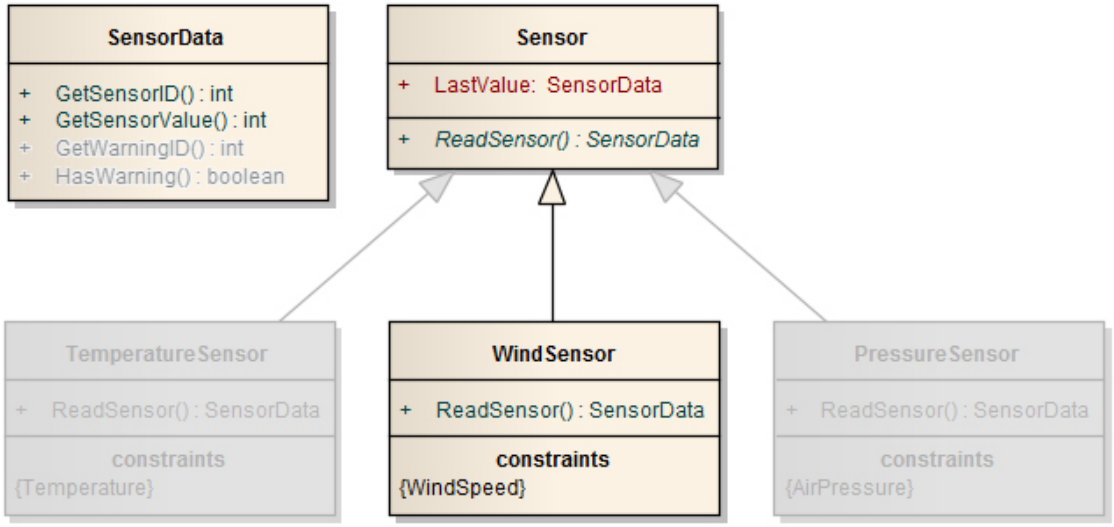
To visualize which model elements would be deleted when deriving a project, we suggest to *lowlight* or de-emphasize these elements (cf. Section 2.5.2). We propose to use the visual variable *value* to accomplish lowlighting. Since *value* is perceived preattentively (Ware [War04], p.166), viewers can distinguish between included and excluded model elements without examining every element individually. Employing lowlighting defines a clear processing order, in which viewers process lowlighted elements last (Moody [Moo07]). Hence, they can examine the master project in the same way as a derived project, with the advantage of being able to see excluded elements. In a derived project, viewers would have to remember the master project's elements to know which elements are missing. Thus, errors, such as falsely excluded elements, are better to spot with lowlighting visualization than without it.

Lowlighting a diagram element can be done by desaturating and brightening colors as described in Section 3.2.2. We show examples of such a lowlighting visualization in Figure 3.2. Figure 3.2(b) displays a configuration, in which only the features *Sensors* and *WindSpeed* are selected. Thus, the two operations of the *SensorData* class related to *Warnings*, are lowlighted in both (a) and (c). So are the classes *TemperatureSensor* and *PressureSensor* plus their related connectors, since the features *Temperature* and *AirPressure* are not selected.





(a) Lowlighting in the model tree. (b) Configuration for the Weather Station Example.



(c) Lowlighting in a class diagram.

Figure 3.2: An example for lowlighting all excluded elements, in the Weather Station's model tree (a) and in its class diagram (c), based on the configuration shown in (b).

Additional to lowlighting excluded elements for a certain configuration, we suggest to *highlight* or emphasize all elements containing variability information that would be *included* in a derived project. Thus, when this visualization is combined with the lowlighting of excluded elements, all elements containing variability information are marked. Since both hue and value are perceived preattentively, they can be recognized easily, which supports users in finding errors. Because they can identify model elements containing variability information at first glance, users can remove variability information where it is inappropriate.

We illustrate an example for highlighting combined with lowlighting in Figure 3.3. Like the lowlighting example, it is based on the configuration shown in Figure 3.2(b). Thus, only *WindSensor* is highlighted. As highlight color, we choose green, because it symbolizes positive things, such as "life" or "go", by common convention (Ware [War04], p.135). This applies well to our visualization, since highlighted elements "stay alive" in derived projects. Color conventions are different for some cultural domains. For example, in China red means life, and green symbolizes death (Ware [War04], p.135). However, there exists no possible compromise between these conventions, so we choose green for highlighting. When making our extension available in different cultures, this conflict should be kept in mind. It could be solved by letting users choose between different colors or color sets.

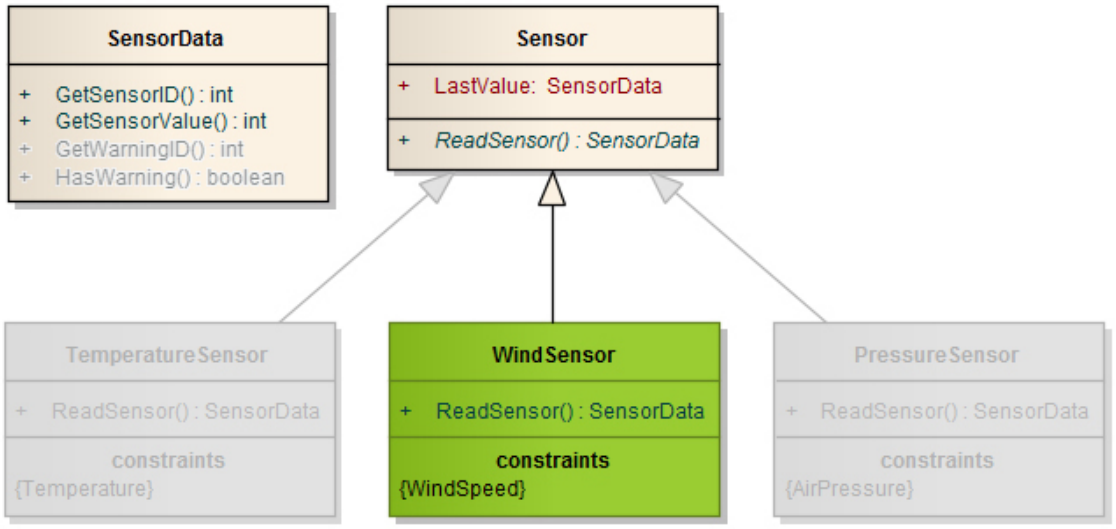


Figure 3.3: An example for lowlighting excluded and highlighting all included diagram elements containing variability information, based on the configuration of Figure 3.2(b).

When highlighting an element, we decide to color only the background and border color; The foreground color stays unchanged, which often means black. This supports the discriminability of the foreground's text. On the other hand, when lowlighting an element, its foreground should also be de-emphasized, since it is more important to perceive an element as lowlighted, than being able to read the foreground's text.

Since variability information applies to a constrained model element and also to its embedded elements (children), it is necessary to lowlight all children of an excluded model element. Moreover, we decide to use this approach also for the highlighting visualization, which aims at identifying all elements related to features. Since children of highlighted elements are related to features by their parent's variability information, they should be highlighted, too.

In Figure 3.4, we illustrate the concepts for highlighting embedded elements. It shows the statechart of Figure 2.6(a) with an additional state containing all other diagram elements. In the visualized configuration, the feature *Warnings* is selected. Since *Parent State* is highlighted, all embedded elements should be highlighted, too. Indeed, this is the case for all diagram elements, except for the constraint "NOT(Warnings)" and its constrained transition. The constraint is not highlighted, because *Warnings* is selected, and thus the constrained elements would be deleted in a derived project even if its parent element were included in the derived project. However, if *Warnings* were deselected in the visualized configuration, all diagram elements would be lowlighted, because all children of *Parent State* would be deleted, regardless of embedded constraints.

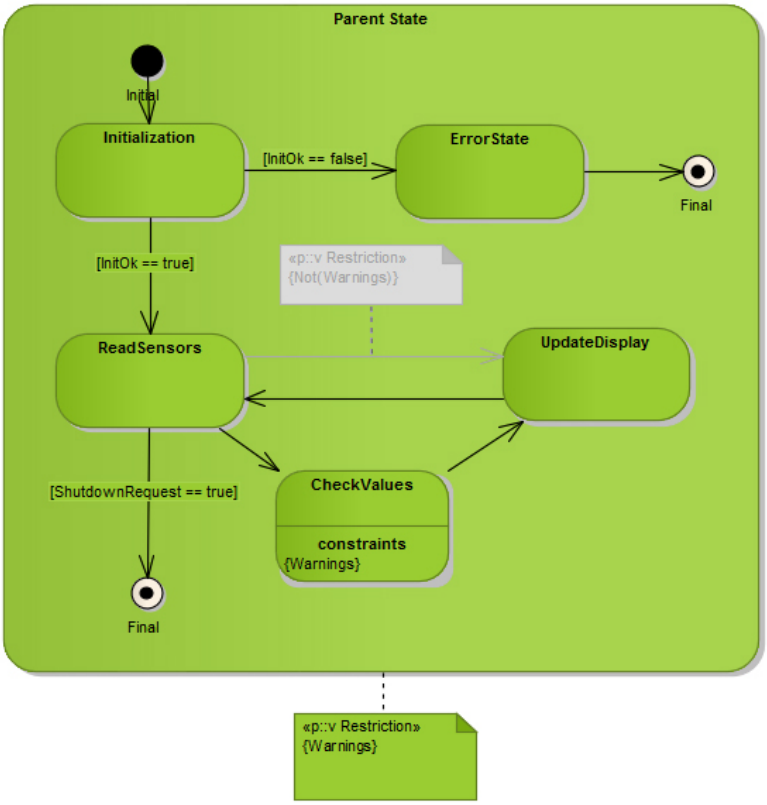


Figure 3.4: An example for highlighting embedded model elements.

3.2.3.2 Visualizing Errors in Variability Information

When using pure::variants and its UML extensions to develop UML projects as a product line, errors in the master project's variability information can only be found by examining the pvSCL expression of each pure::variants constraint. This is very time-consuming and error-prone, since errors – such as a missing letter – can easily be overlooked. Therefore, we argue that highlighting elements containing an error can increase the efficiency of the error-finding process significantly. In order to find these errors automatically, we propose to check the pvSCL syntax of all pure::variants constraints and highlight all diagram elements that contain constraints with errors. We choose red as visualization color, since it symbolizes danger by common convention (Ware [War04], p.135).

Not only syntactic errors can occur and lead to wrong results when deriving a project, but also semantic errors, such as a misspelled feature in a pvSCL expression. Therefore, we propose to check whether the parameters of a pvSCL expression are contained in the feature model of the configuration. If they are not, the expression is still syntactically correct, but it does probably not lead to the desired results when deriving a project. We suggest to highlight all semantic errors using a light yellow highlight color. We choose this color, since it is distinct from red and green and symbolizes caution (Braun et al. [BMS95]).

In Figure 3.5, we demonstrate an example for highlighting syntactic and semantic errors. *TemperatureSensor* is highlighted in red, because it contains a pure::variants constraint with the pvSCL expression "Temperature AND". This contains a syntactic error, since "AND" needs two arguments to produce a syntactically correct pvSCL expression (e.g., "Temperature AND WindSpeed". *WindSensor* contains a constraint with a semantic error, because the feature "Wind<u>s</u>peed" does not exist. Hence, it is highlighted in yellow.

The two presented visualizations enable users to find most errors of the variability information at first glance. Thus, the process of finding and correcting errors becomes more efficient. But, why do we have to search for errors in the first place? Would it not be more appropriate to prevent errors in variability information when adding this variability information? Next, we discuss this question, and propose solutions for supporting users in creating and editing variability information.

3.3 Editing Variability Information

Adding variability information to a master project when using pure::variants is quite simple: A modeler adds contraints to elements related to features and enters a pvSCL expression as the constraint's text. This pvSCL expression describes how the element is related to the feature model of the product line. The problem of this process is that modelers enter pvSCL expressions without any support, which we already stated in the third usability issue of Section 3.1. This means that users have to know the exact spelling of feature names, and probably have to switch between pure::variants and the UML tool a lot to look at the feature model. This contradicts heuristic *recognition*

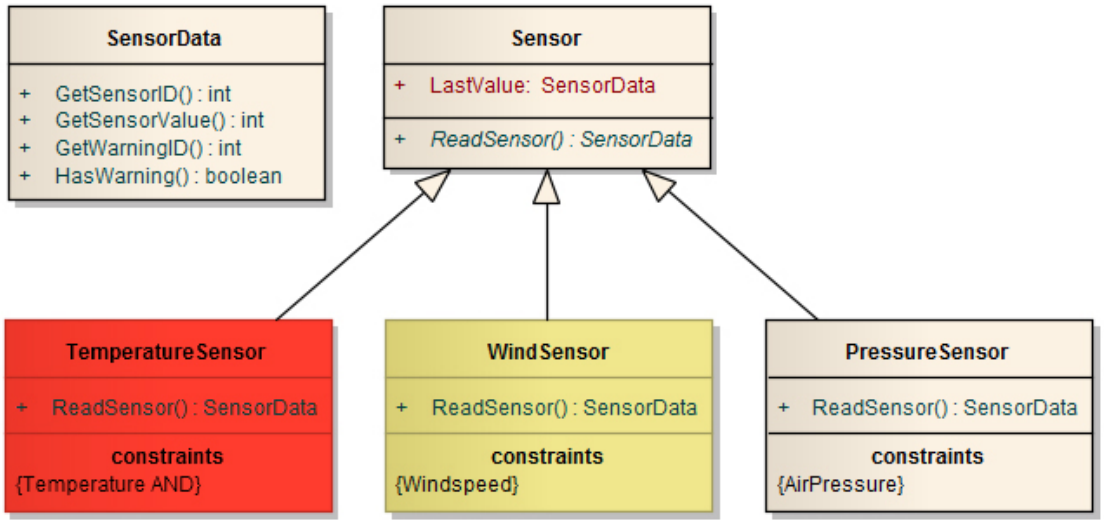


Figure 3.5: Example for highlighting syntactic and semantic errors in the weather station's class diagram.

rather than recall and heuristic *flexibility and efficiency of use*. Furthermore, small errors, such as typing errors, could stay unnoticed, because no automatic check for errors is implemented. This conflicts with heuristic *error prevention*.

Hence, there are three tasks to improve efficiency and reduce errors: (1) Assisting modelers in finding possible feature names and spelling them correctly, (2) supporting modelers in understanding the entered pvSCL expression, and (3) notifying modelers, whether the expression contains errors.

To address these tasks, we suggest to provide UML tools with a *constraint editor*. This editor should be capable of syntax highlighting to support understanding of the written expression, and autocompletion with proposal of all possible completions. By using autocompletion, necessary keyboard input and errors are reduced (Preim and Dachselt [PD10], p.326,330): Typing errors can be eliminated almost completely, because users can hit CTRL + space, and select fitting feature names from a list of available features. Since they do not have to remember the exact spelling of features, they also do not have to switch between pure::variants and the UML tool to look at the feature model. This also increases efficiency, apart from the reduced keyboard input resulting of autocompletion. Hence, we solve the third usability issue specified in Section 3.1.

Since our implementation is based on pure::variants and its UML extensions, we propose a design similar to pure::variants' constraint editor, shown in Figure 3.6. Thus, the colors used for syntax highlighting and the icons used for proposing completions should be the same as in pure::variants' constraint editor.

Additional to the functions of pure::variant's constraint editor, we suggest to check for syntax and semantic errors, before applying a constraint to a UML project. Thus, we

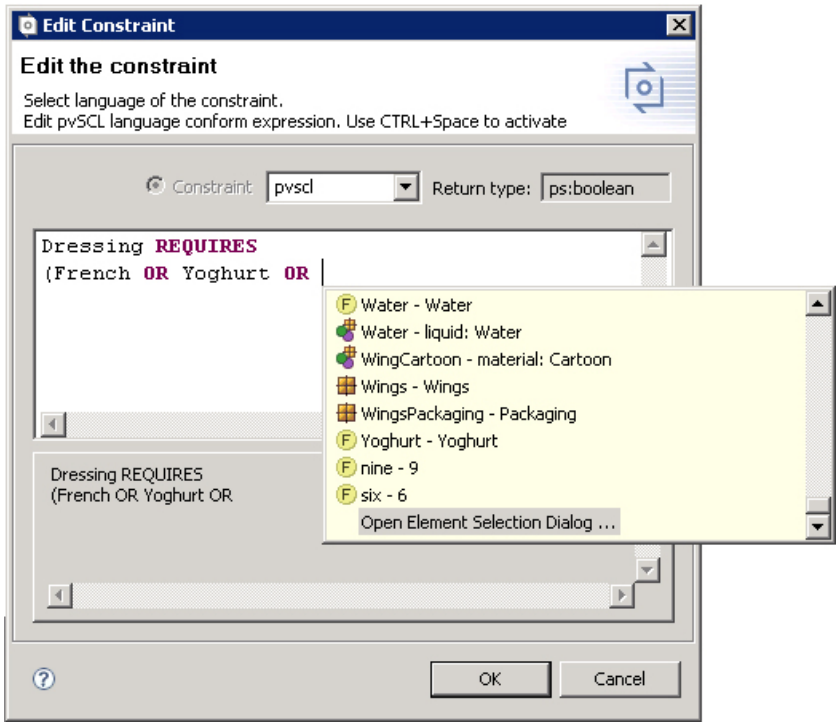


Figure 3.6: The pure::variants constraint editor ([pur], p.78)

address the fourth usability issue specified in Section 3.1, which states that Architect and Rhapsody do not report errors in pvSCL expressions. Users should be notified of the error's type (syntactic or semantic) and of its position in the pvSCL expression. This enables them to correct it more efficiently, because no search for the error's location is necessary. We propose to underline pvSCL elements containing errors in red, which is consistent with the highlighting of model elements containing errors (cf. Section 3.2.3.2).

3.4 User Interface

Having described which functions our proposed UML tool extension should have, it remains to present a UI that allows users to interact with our extension. Since it is a tool extension, our UI should be embedded as seamlessly as possible into the tool's UI to be consistent with the UI (cf. Section 2.5.1.2, usability heuristic *consistency and standards*). In order to achieve this, we suggest to add a toolpane containing our UI.

We present the design of the proposed UI in Figure 3.7. In the remainder of this section, we explain which functionality the depicted UI elements have, and our reasons for using them. We use the displayed labels to refer to UI elements.

Since our concepts include a visualization for configurations, we need to enable users to load configurations from a file, show the name of the loaded file, and give them an overview of its contents. Thus, we provide a button displaying an open dialog when clicked (see label (1) in Figure 3.7) and a label showing the name of the current

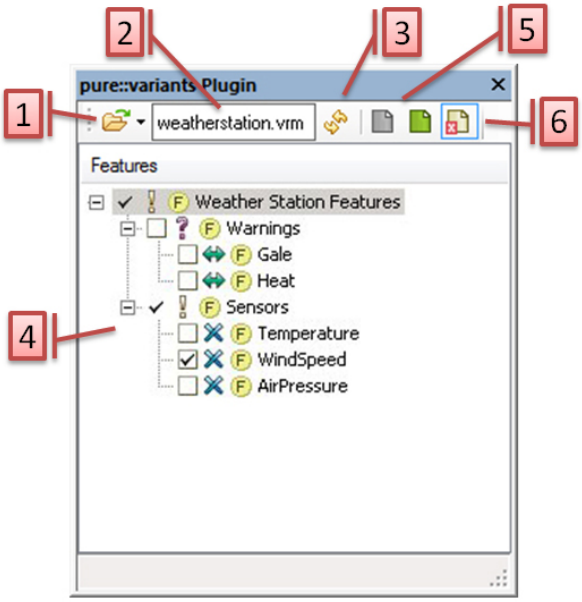


Figure 3.7: Proposal of a user interface for our UML extension.

configuration (2). We add them to a toolbar grouping all UI elements for accessing the extensions functions. To support users in switching between configurations, we suggest to use a split button (1) that enables users to expand a dropdown menu. This menu lists previously loaded configurations. When a user selects a configuration from it, the selected configuration is loaded. Furthermore, users might want to adapt configurations while modeling the master project. Thus, a way of reloading an opened configuration is necessary. We suggest to use another button to achieve this (3). We take the icons for the two proposed buttons (1)(3) from pure::variants, so that users are familiar with them. This supports consistency with pure::variants, which is one of the usability heuristics of Section 2.5.1.2.

We propose to use the pure::variants tree layout to give an overview of the configuration's contents (4), because our implemention is based on pure::variants. Hence, users do not have to learn a new notation for displaying configurations, since they are already familiar with the pure::variants tree layout. Thus, we improve the learnability of our extension.

The presented UI elements enable users to load and view configurations. The next step is to let users apply the highlighting, lowlighting, and error visualizations presented in Section 3.2. To achieve this, we suggest to use toggle buttons (5), which initiate the corresponding visualization when selected. Label (6) indicates the toggle button for highlighting errors. It is selected, thus error visualizations should be applied to all open diagrams and the project's model tree. For consistency, we create icons of a similar design as pure::variants' icons. An advantage of using toggle buttons is that they are easy to understand and users can combine all visualizations selectively. This means, for example, that lowlighting and highlighting of configurations can be combined with

highlighting of errors, which supports flexibility (cf. Section 2.5.1.2, heuristics *flexibility and efficiency of use*). Hence, one model element could be lowlighted, while at the same time containing an error. Which visualization technique should be used in this case? We solve this conflict by proposing a hierarchy of visualization techniques ordered by importance. This means, if a visualization technique is more important than another technique, and if both techniques apply to the same model element, the most important technique is preferred. We have decided to use the following hierarchy: (1) Syntactic errors, (2) semantic errors, (3) lowlighting of excluded elements for a configuration and (4) highlighting of included elements.

Since errors affect the outcome of project derivations, and thus also configuration visualizations, we consider them most important. Of the two error types, we regard syntactic errors as more important than semantic errors, because semantic errors still represent a correct pvSCL expression. Configuration-based lowlighting overrides highlighting of included elements, since an excluded element cannot be included in a derived project (cf. Figure 3.4).

Apart from opening variant models, it is necessary to load one or more feature models. Therefore, we suggest to provide a *preferences dialog* allowing users to enter paths to multiple feature models. To support users working with different computers on the same project, we also propose to treat these paths as relative paths denoting only the relative location of a feature model to the UML project. Hence, when all necessary models of a project are contained in one folder, and a users saves this project to one computer and opens it on another, the feature models could be found by the extension, even though the absolute path to them has changed.

3.5 Summary

This concludes the concepts for a UML tool extension. In this chapter, we addressed the first two goals set in Chapter 1: We identified usability issues of Architect and Rhapsody when used for SPLE with pure::variants by pure-systems.

We list the issues again and summarize how we solved them:

1. Users have to switch between pure::variants and the UML tool to view a derived UML project, which conflicts with heuristic *flexibility and efficiency of use*.

To solve the first issue, we proposed lowlighting and highlighting visualizations to preview a configuration, which improves the efficiency of modeling a master project.

2. Existing errors can only be found by scanning each pvSCL expression individually, which also contradicts heuristic *flexibility and efficiency of use*.

In order to address this issue, we suggested error visualizations of syntax and semantic errors.

3. Users have to enter pvSCL expressions without support, which conflicts with heuristic *error prevention* and heuristic *recognition rather than recall*.

To solve the third issue, we presented a constraint editor that supports autocompletion, and syntax highlighting.

4. Errors in pvSCL expressions are not reported. This contradicts usability heuristic *error recognition and recovery*.

Therefore, we proposed to add error detection to the constraint editor. Hence, we solve all identified issues. To enable users to work with the proposed extension, we also made proposals for a UI. Next, we present our implementation of the concepts devised in this chapter.

4. Implementation

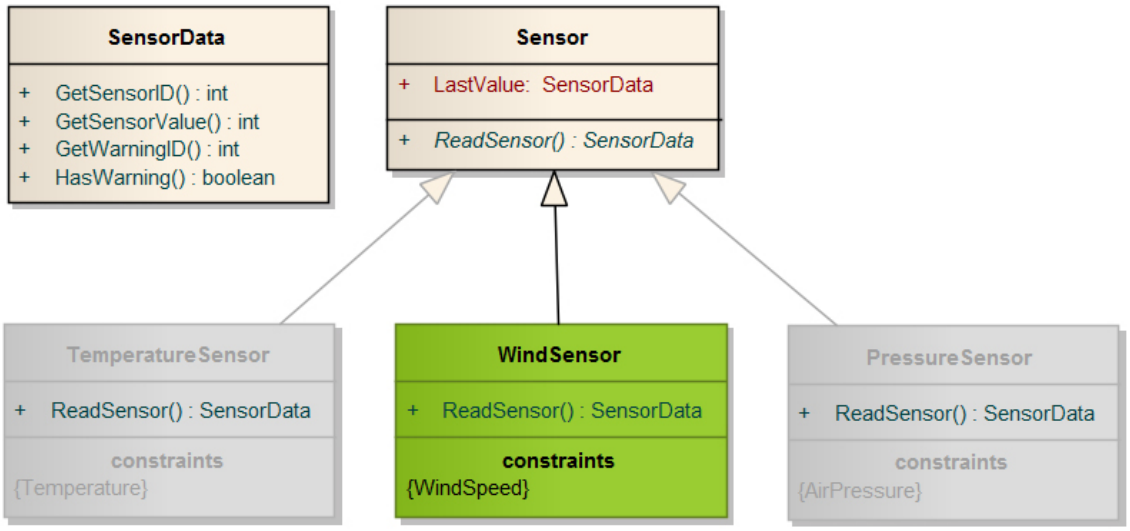
In this chapter, we present an implementation of the concepts proposed in the previous chapter. To test whether the suggested concepts can be realized, we extend two different UML tools. We choose to extend IBM Rational Rhapsody (Rhapsody) and Enterprise Architect (Architect) – tools on which pure::variants' UML extensions are based. Thus, we support users of pure::variants' UML extensions in modeling a master project.

We structure this chapter similar to the previous chapter. The concepts of the last chapter are merely based on what would produce good usability results, but not on what is possible. Therefore, we analyze in this chapter which proposed features could be realized and which could not. In each section, we describe what we realized, what we changed, what we could not implement and the reasons for our decisions. First, we describe our implementation of the visualizations proposed in Section 3.2.

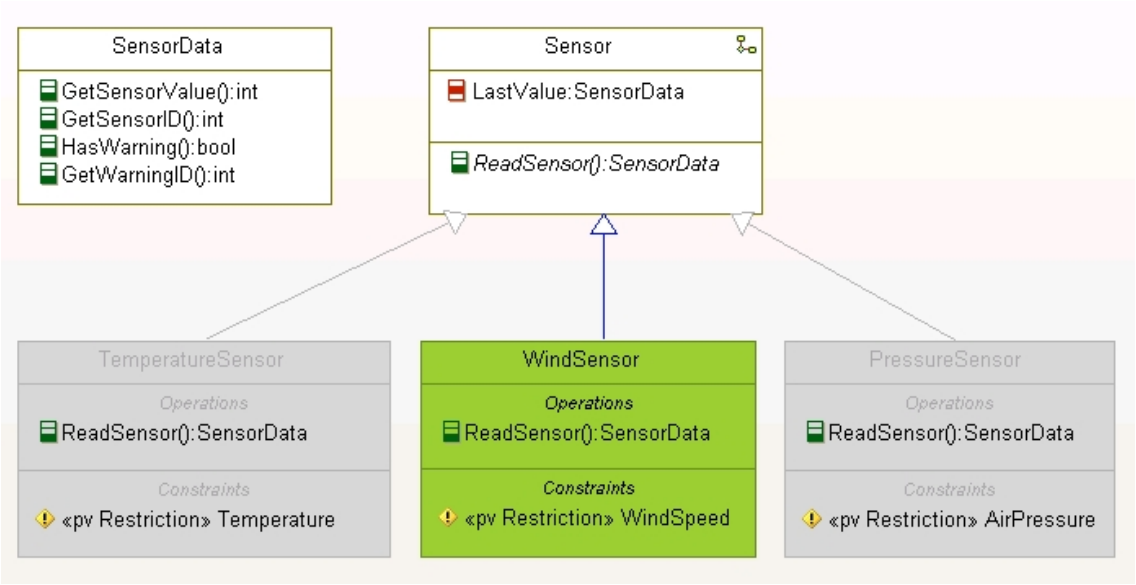
4.1 Visualizations

Apart from some details, we succeed in implementing the visualizations proposed for the diagram view (cf. Figure 2.8, label 7). On the other hand, both UML tools do not allow manipulation of the model tree in their current versions. Thus, we cannot apply visualizations to it. This is a major drawback, because visualizations in the model tree give an overview of the whole project and its hierarchy, whereas visualizations in diagrams show often only a small part of the project.

In Figure 4.1, we illustrate our implementation of both configuration visualizations in Architect (a) and Rhapsody (b). The shown result is almost consistent with our concepts displayed in Figure 3.3. The only difference is that the operations HasWarning(), GetWarningID() and ReadSensors() are not lowlighted, because both tools do not allow coloring of operations and attributes. The same applies to contraints displayed in classes in Rhapsody (e.g., "Temperature" or "AirPressure" in Figure 4.1(b)). This is another major drawback, since the resulting visualization is not completely consistent.



(a) Architect



(b) Rhapsody

Figure 4.1: Configuration visualizations in Architect (a) and Rhapsody (b), based on the configuration shown in Figure 3.2(b).

A solution to this problem could be to change the background color of the containing element, or change its texture. We do not choose to do so, because it would complicate the simple design of the visualizations. Since we provide no workaround for this issue, it must be communicated to users, so they know the visualizations do not cover attributes and operations (and Rhapsody constraints represented inside a class).

The same problems apply to the otherwise correct error visualizations displayed in Figure 4.2. If the constraints of GetWarningID() or HasWarnings() contained errors, the operations would not be colored. A small difference of our Rhapsody implementation to the proposed concepts of Section 3.2.3.2 is that elements with semantic errors are highlighted in yellow with a red border instead of a yellow one (cf. Figure 4.2). This is the case, because in Rhapsody the default color of constraints is similar to the proposed yellow. Thus, we provide a red border for better discrimination of constraints with or without semantic errors. For error visualizations in Architect see Figure 3.5.

Another tool restriction occurs in Architect: the elements of sequence diagrams cannot be colored correctly. Sequence diagrams display an interaction as a two dimensional chart (Rumbaugh et al. [RJB04], p.87). This tool restriction must also be communicated to users. Next, we present our implementations of the constraint editor.

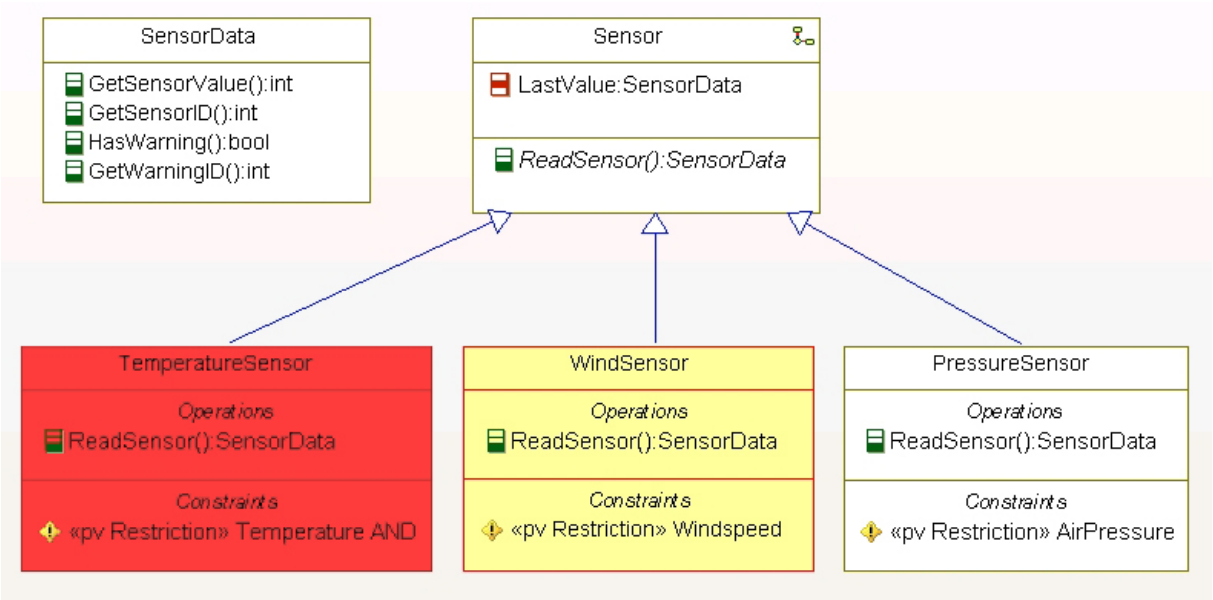
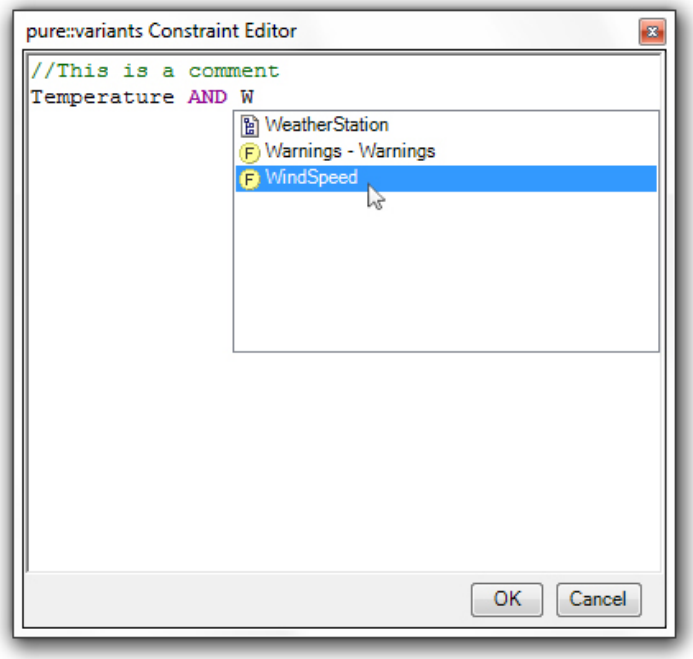


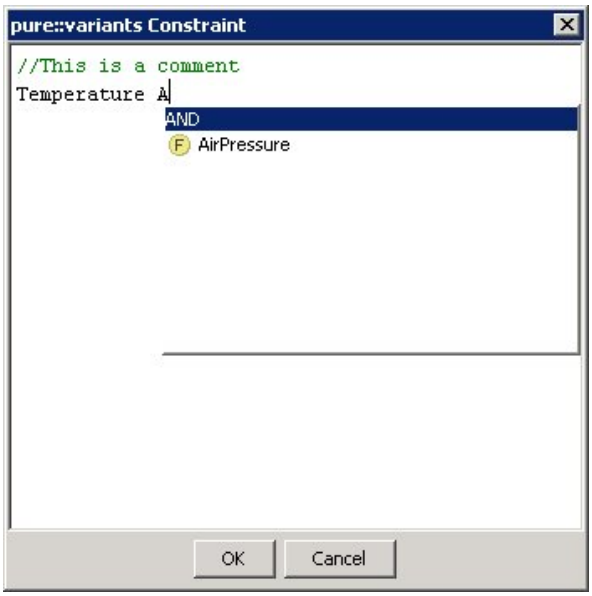
Figure 4.2: Error visualization in Rhapsody.

4.2 Constraint Editor

We implement the constraint editor in a simplified form of the pure::variants constraint editor. In Figure 4.3, we illustrate the constraint editors of both tools. The figures show our implementation of autocompletion and syntax highlighting.



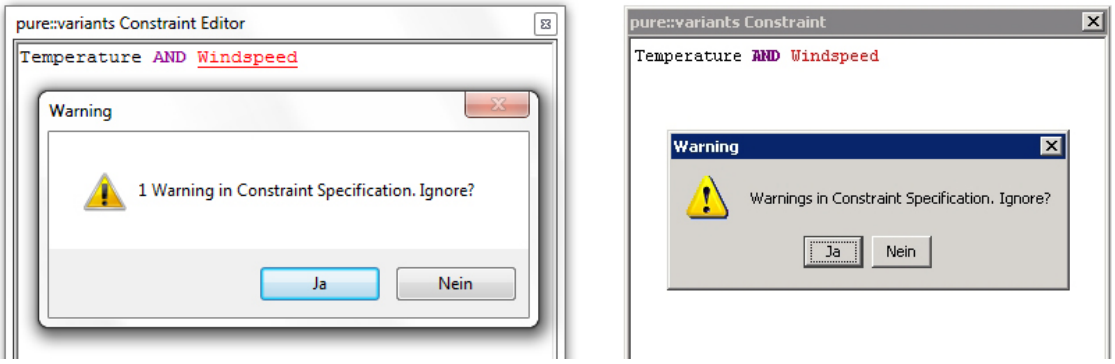
(a) Architect



(b) Rhapsody

Figure 4.3: The constraint editor implemented for Architect (a) and Rhapsody (b).

Both tools do not restrict us in our implementation of the constraint editor. Thus, we realize all proposed concepts. This includes checking a pvSCL expression for syntax and semantic errors. We illustrate an example of such a semantic error in Figure 4.5. The dialog of the example does not refer to a semantic error, but to a warning, because pvSCL expressions with semantic errors are still syntactically correct. The pvSCL parameter "Windspeed" contains a typing error, since the correct spelling would be "WindSpeed". Hence, it is highlighted in red. For brevity, we omit examples for a syntax error message. They are similar to the shown example.



(a) Architect (b) Rhapsody

Figure 4.4: Semantic error message of the constraint editor with highlighting of errors in Architect (a) and Rhapsody (b).

Apart from the constraint editor itself, it is also important to provide an easy way to open it. We do this by extending the context menus of elements with options for adding new pure::variants constraints, converting usual constraints to pure::variants constraints, and editing existing pure::variants constraints. These options only appear if they apply to the selected element. For example, the option to edit existing constraints only appears for constraints with a pure::variants stereotype. When clicking the option, our extension checks the constraint's text for errors, the constraint editor opens showing the text, and if it contains errors, they are highlighted. For efficiency, it is also possible to select multiple elements, and add the same constraint to all of them.

In Architect, it is not possible to open a context menu for a constraint that does not have an individual diagram element, such as the "Temperature" constraint of class *TemperatureSensor* in Figure 4.1. Hence, these constraints could not be edited using the constraint editor. Therefore, we enable users to access these constraints through the context menu of the containing element (e.g., *TemperatureSensor*). In Figure 4.5(a), we demonstrate how users can access the editor for all pure::variants constraints contained in *TemperatureSensor* and *PressureSensor*. It also shows options to add new or delete existing pure::variants constraints.

Since the context menus of all Rhapsody constraints can be accessed through the model tree, and because Rhapsody does not allow hierarchical menu entries, we do not provide

Rhapsody model elements with options to edit contained pure::variants constraints, or delete these constraints. In Figure 4.5(b), we illustrate how to add a pure::variants constraint to multiple selected classes.

4.3 User Interface

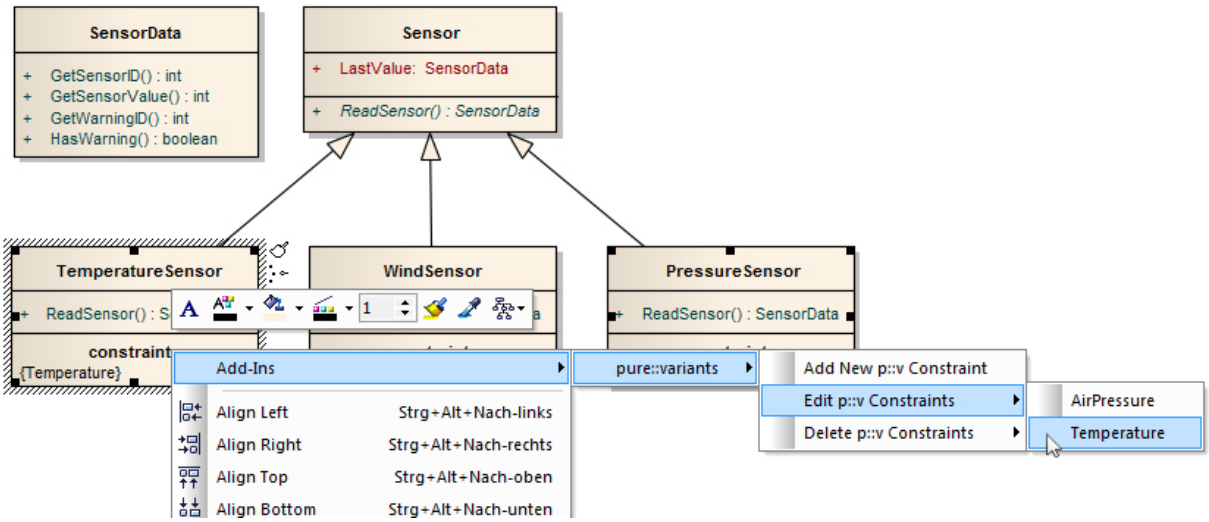
In Section 3.4, we stated that the UI of our extensions should be integrated seamlessly into the UML tool's UI. We are able to realize this only for the Architect extension, but not for Rhapsody, which does not provide a way to extend its UI with a custom UI. Therefore, we implement the UI of the Rhapsody extension based on a Java window that stays always on top of other windows. This is necessary, because it should be visible even when Rhapsody has the user's focus. The solution is not ideal, since the Java window is independent and therefore does not adapt to Rhapsody's UI. For consistency, we implement the UI of Rhapsody's extension such that it takes the look and feel of the user's operating system.

In Figure 4.6, we illustrate both UI implementations. They are similar to the concepts of Figure 3.7 in Section 3.4. Therefore, we only indicate differences to the proposed concepts.

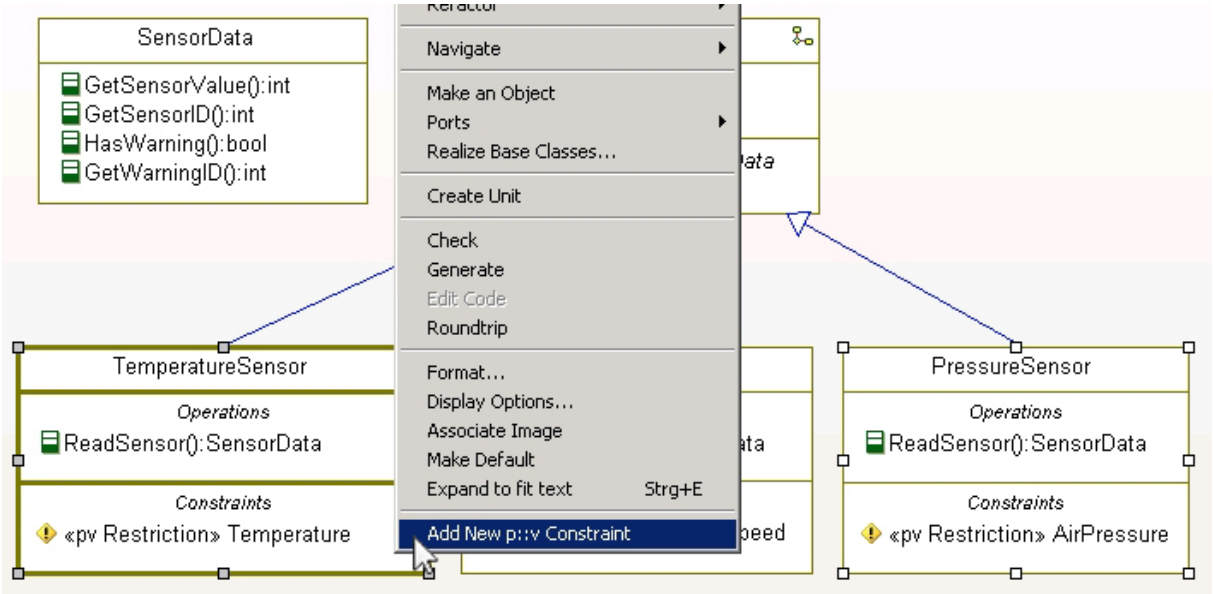
Unlike our suggestion to use the pure::variants tree layout for displaying the loaded configuration's content, we use a simple list with checkboxes indicating whether the corresponding feature is checked. To be able to use the pure::variants tree layout, it would have to be reimplemented, because its implementation is integrated into Eclipse IDE and cannot be used outside Eclipse. Since reimplementation is very time-consuming, yet does not introduce anything new, we show the features in a simple list instead. To support users in relating to the pure::variants view, we sort them in the same order as in pure::variants. Furthermore, we indicate which configuration visualizations are enabled by coloring the features. Selected features are highlighted in green, if the visualization for included elements is enabled. Deselected features are lowlighted, if the visualization for excluded elements is enabled. Neither highlighting nor lowlighting is activated in Figure 4.6 to have a simple overview. By coloring the feature list, we always provide feedback (additional to the actual visualization) such that users can see which types of configuration visualizations are enabled, as illustrated in Figure 4.7.

We also show another way to provide feedback in Figure Figure 4.7. Other than in our Rhapsody extension, the Architect extension contains a progress indicator at the bottom of the UI that visualizes the remaining time until a visualization is applied. This feature should also be added to our Rhapsody extension in future versions, since applying models may take longer than 10 seconds for huge projects (cf. Section 2.5.1.2, heuristic *visibility of system status*).

Figure 4.6(b) differs from Figure 4.6(a) in two aspects: It shows a tooltip next to the refresh button, and it displays one more button next to the visualization toggle buttons. The tooltip helps users in understanding the functionality of UI elements, such as buttons. For this purpose, we provide tooltips for every button in our Architect

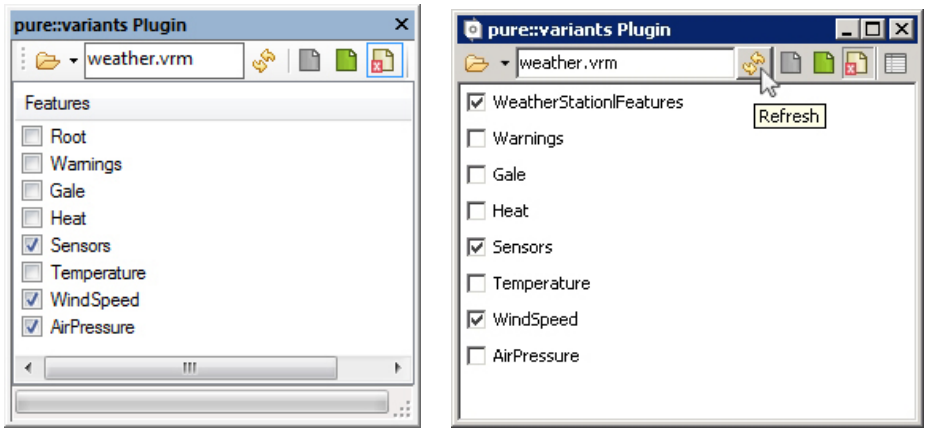


(a) Architect



(b) Rhapsody

Figure 4.5: Using Architect's context menu to *edit existing* (a), and Rhapsody's context menu to *add new* pure::variants constraints to selected classes (b).



(a) Architect (b) Rhapsody

Figure 4.6: UIs of the Architect (a) and Rhapsody (b) extension.

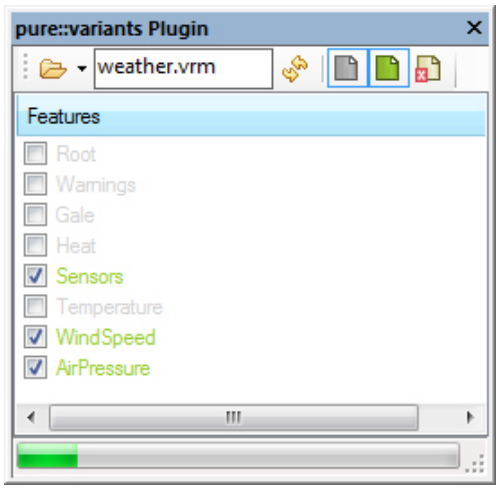
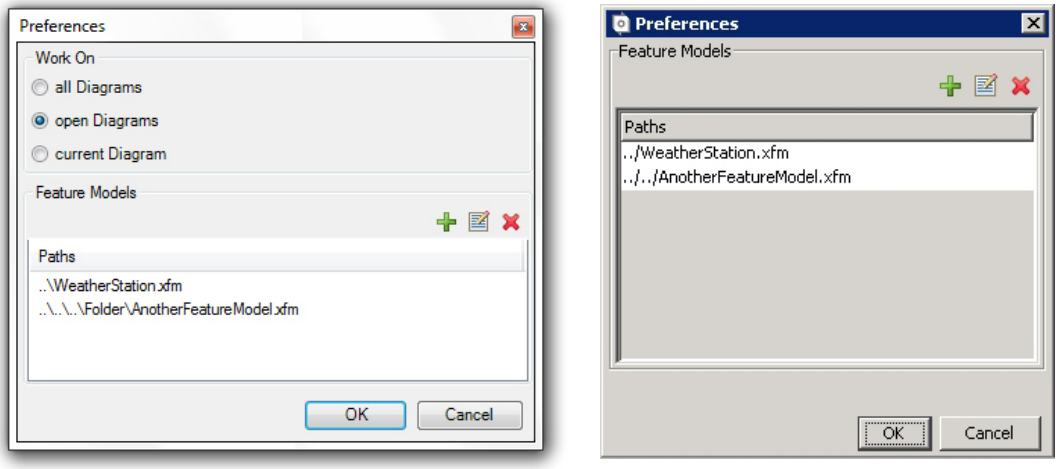


Figure 4.7: UI of the Architect extension when both configuration visualizations are enabled.

and Rhapsody extensions. When a user selects the additional button, a preferences dialog opens, based on the concepts of Section 3.4. In Architect, we did not implement this button, since the preferences can be reached through Architect's menubar, which is not possible in Rhapsody.

In Figure 4.8, we illustrate our implementations of the preferences dialog. Both provide a means to select paths to feature models as described in Section 3.4. Users can enter the feature model paths into a list using the "+" button. When users click the button, an open dialog is shown that helps in selecting the path. With the other buttons, they can edit or delete the selected path, which is represented as a relative path. They can also edit the list entries manually, which supports flexibility (cf. Section 2.5.1.2, heuristic *flexibility and efficiency of use*). Additional to the feature model selector, Figure 4.8(a) shows a section "Work on" with the options "all Diagrams", "open Diagrams", and "current Diagram". We add this option to give users the opportunity of adapting the performance to their needs. In huge UML projects, the processing of a visualization can take very long. Therefore, users can choose, for example, to process only the current diagram, which is significantly faster, but does not update when switching diagrams. This provides a compromise between performance and consistency of applying visualizations.



(a) Architect (b) Rhapsody

Figure 4.8: Preferences dialogs of Architect (a) and Rhapsody (b).

4.4 Summary

In this chapter, we addressed the third goal we defined in Chapter 1: We described our implementation of the extensions to Rhapsody and Architect proposed in Chapter 3. During implementation of the suggested concepts, we encountered several tool restrictions, such as the inability to integrate our extension into Rhapsody's UI, or to apply

visualizations to the model tree, to attributes and operations, to Rhapsody constraints represented inside a class, or to Architect's sequence diagrams.

Besides the encountered restrictions, we implemented the proposed concepts. This concludes the description of our extensions. Next, we evaluate their usability.

5. Evaluation

In this chapter, we evaluate whether the extensions presented in the previous chapter improve the usability of Architect and Rhapsody when used for SPLE with pure::variants. For this evaluation, the best method would be user studies (Nielsen [Nie94b], p.165). For example, we could conduct a user study for Architect in two sessions: In one session users that are familiar with pure::variants' Architect extension would have to create a master project without our extension. In the other session, they could use our extension for the same task. Both times, we would measure the usability attributes introduced in Section 2.5.1.1 (e.g., by measuring the time to finish the task, we could measure the efficiency, or by counting errors in pvSCL expressions, we could measure the error rate). To determine whether our extensions improve Architect's usability, the results could be compared.

However, we could not conduct a user study, because of time constraints, and since appropriate subjects are hard to find for pure::variants' UML extensions. Instead, we employ the usability heuristics introduced in Section 2.5.1.2. For each heuristic, we reason whether our extension improves the heuristic or not. Thus, we cannot measure *how much* the usability has improved, but we can argue *whether* it has improved and why. For better overview, we present the usability heuristics again:

1. *Visibility of system status*: A system should always inform users about what it is doing. Each user input should be followed by a reaction of the system.

2. *Match between system and real world*: A system should employ the users' terminology to communicate with them, rather than using its internal terminology and make use of users' background knowledge.

3. *User control and freedom*: Users should always feel in control of the system they are using. For example, they should always be able to undo or redo an action, close an open dialog, or interrupt a process that takes too long.

4. *Consistency and standards*: An interface should be consistent, which means that required actions should be consistent in similar situations. Terminology in prompts, menus and help screens should be identical, and consistent color, layout, fonts etc. should be used.

5. *Error prevention*: UI elements should be chosen such that the error rate is kept at a minimum.

6. *Recognition rather than recall*: Users are much better at recognizing something than at remembering it without help. Hence, a system should rely on UI elements letting users choose from a list of possible items, or letting users modify data rather than entering it completely from scratch.

7. *Flexibility and efficiency of use*: A system should be flexible and efficient to use. This can be achieved, for example, by providing shortcuts to frequently used functionality.

8. *Aesthetic and minimalistic design*: The design of a system should be aesthetic, so that users like using the system. Since the number of information pieces processed in short-term memory is limited to seven plus or minus two units (Miller [Mil56]) displays and the system's design should be kept as simple as possible.

9. *Error recognition and recovery*: When an error occurs, the system should detect the error immediately and offer simple, constructive, and specific instructions for recovery.

Since we aim to *improve* the usability of UML tools used for SPLE, we compare how well each heuristic is fulfilled when modeling a master project *with* or *without* our extensions. When a heuristic is better satisfied by the UML tools *with* extension, we conclude that the usability has improved concerning this heuristic. We use the results of all heuristics to evaluate the overall improvement of the UML tools' usability.

Visibility of System Status

Our UML tool extensions show their current status in several ways: For example, the state of the toggle buttons shows which visualizations are currently selected. Furthermore, the feature list (cf. Figure 4.7) illustrates which configuration visualizations are enabled, by highlighting and lowlighting the affected features. In the constraint editor the list of proposals is updated while the user types, giving instant feedback on possible elements of the pvSCL expression. In Architect, we additionally implement a progress indicator, providing users with feedback on how much time remains until the model is updated. Thus, the Architect extension provides better feedback than the Rhapsody extension.

Based on these statements, we reason that our extensions satisfy heuristic *visibility of system status*.

Match between System and Real World

Since users of our extensions also use pure::variants, we use its terminology. For example, users can load "Variant Models" and "Feature Models" like in pure::variants. We also rely on users' background knowledge about variants and feature models. Hence, we satisfy heuristic *match between system and real world*.

User Control and Freedom

According to heuristic *user control and freedom*, all dialogs of our implementations can be closed at any time, and visualizations can easily be reverted by deselecting the visualization's toggle button. On the other hand, we provide no undo or redo action when a constraint is added, edited, or deleted using our context menu triggers. We omit the undo/redo action, because we could only have implemented it inconsistently with the UML tools' UI, since we could not use the undo/redo menu entry of Rhapsody or Architect for our extensions.

Thus, heuristic *user control and freedom* has not improved compared to the UML tool without our extension. We satisfy it only for some aspects.

Consistency and Standards

We achieve consistency of our extensions in several ways: We use the same icons, or icons similar to those of pure::variants. Furthermore, we integrate our Architect extension into the UI of Architect using a toolpane. Other extensions can be added to Architect in the same way. Hence, accessing our extension is consistent with accessing other extensions that might be installed. Since integration into Rhapsody's UI is currently not possible, our Architect extension is more consistent than the Rhapsody extension. On the other hand, the look and feel of both extensions is consistent with the look and feel of the user's operating system. Thus, users of both implementations are familiar with the used UI elements. Furthermore, our visualizations can only be applied inconsistently to some extent, because both Rhapsody and Architect do not allow coloring of the model tree (cf. Figure 2.8, label 6), or of attributes and operations. Moreover, visualizations cannot be applied correctly to UML sequence diagrams (cf. Section 4.1).

Based on these statements, we reason that our extensions satisfy heuristic *consistency and standards* with some exceptions.

Error Prevention and Recognition Rather than Recall

One of the most important steps in developing a master project is adding and editing of constraints that contain variability information. Without our extensions, users are not supported in typing pvSCL expressions, which contradicts heuristics *error prevention* and *recognition rather than recall*. Thus, errors, such as typing errors or semantic errors, may occur when users incorrectly remember the name of a feature.

To prevent these errors, our extensions include a constraint editor with autocompletion and syntax highlighting (cf. Section 4.2). Autocompletion reduces typing errors, because it completes words always correctly. Furthermore, users can select possible words from a list of proposals (cf. Figure 4.3). Since users are much better at recognizing something than at remembering it without help, our extension reduces semantic errors by providing proposals (Nielsen [Nie94b], p.129). Moreover, we assume that syntax highlighting also reduces errors, because users expect keywords to be highlighted after entering them. When a keyword is not highlighted, it is obvious that the entered word is not a keyword and the user can correct the error.

Since our extensions provide error prevention methods, whereas Architect and Rhapsody do not, we state that they considerably improve heuristic *error prevention*. To achieve error prevention, our extensions use methods of recognition rather than recall, which is not the case in Architect and Rhapsody. Hence, we conclude that they also improve heuristic *recognition rather than recall* significantly.

Flexibility and Efficiency of Use

To improve the UML tools' efficiency and flexibility, we use several methods: Firstly, we provide users with visualizations that highlight relevant aspects of diagrams, such as errors or elements that should be deleted. These aspects would otherwise be much harder to find, since users would have to scan the contents of each constraint and process their meaning. In our extension, we use preattentive visual features to mark these aspects in the diagram (cf. Section 3.2). This approach significantly reduces user memory load compared to the approach without our extension, and thus improves the efficiency of the UML tools. Secondly, users can edit constraints of a UML model element through its context menu. This method requires in most cases less user actions than the usual constraint editing process. For example, to add a constraint to an element in Rhapsody, users would have to open the properties dialog of an element, navigate to the "Constraints" tab, enter the constraint's text, specify its type, save the constraint, and press "OK". When using the extension, it suffices to open the element's context menu, select "Add new p::v Constraint", enter the constraint's text and press "OK". Furthermore, this action could be applied to multiple elements at a time, which also improves efficiency of use. Thirdly, we provide users with a shortcut to previously loaded variant models: The open button is a split button that provides a list of previously loaded models, which can be loaded with one click.

Since the UML tools *without* extension do not provide shortcuts or visualizations to speed up the process of modeling a master project, but our extensions do, we conclude that our extensions improve the UML tools efficiency.

Aesthetic and Minimalistic Design

According to heuristic *aesthetic and minimalistic design*, we use minimalistic design in our visualizations: We employ less than five colors, which can be easily distinguished (Rice [Ric91]). How aesthetic the extension's UI elements are, is subjective, so we

do not evaluate it. This could be done in a future user study by asking users for their opinion about our extensions' appearance. If enough users were asked, we could evaluate the aesthetics of our extension by taking the mean value of the results. In any case, the appearance of UI elements depends on the used operating system. Thus, we can influence it only in a limited way.

Hence, we argue that our extensions satisfy heuristic *aesthetic and minimalistic design* only concerning minimalistic design. However, we cannot evaluate it concerning its aesthetics.

Error Recognition and Recovery

When an error cannot be prevented, the system should offer simple, constructive, and specific instructions for recovery (Shneiderman [Shn97], p.75). In the constraint editor, we do this by checking syntax and semantic of the entered pvSCL expression, and displaying a message, which specifies the type of error. If the error is semantic, the faulty words are highlighted in the constraint editor (cf. Figure 4.5). Furthermore, we inform users when a variant model cannot be loaded, or when a feature model cannot be found in the preferences dialog. We also provide the corresponding path.

Thus, error recognition and recovery has improved when editing constraints, because Architect or Rhapsody do not check the content of constraints, but our extensions do.

This concludes the evaluation of usability heuristics. To illustrate our evaluation, we give an overview of the results in Table 5.1. "+" means the extension of the top row improves the heuristic of the leftmost column, compared to the UML tool without extension. "∘" indicates it satisfies the heuristic, whereas "-" means that an aspect of the tool does not fulfill the heuristic. "?" indicates that we could not evaluate part of the heuristic. We restrict the evaluation to actions related to modeling a master project for pure::variants.

The evaluation results show that the usability of both UML tools was improved by our extensions concerning four heuristics: *Error prevention*, *recognition rather than recall*, *flexibility and efficiency of use*, and *error recognition and recovery*. Both UML tools have major usability issues regarding these heuristics (cf. Section 3.1), such as missing error prevention techniques or no support for finding errors. Our extensions fix these issues. They also satisfy the other heuristics (*visibility of system status*, *match between system and real world*, *user control and freedom*, *consistency and standards*, and *aesthetic and minimalistic design*). However, our extensions do not improve these heuristics compared to the UML tools without extension, since the tools have no major usability issues regarding these heuristics.

The results also show that in some aspects the extensions do not fulfill the usability heuristics. For example, our visualizations are not completely consistent, since we

	Architect Extension	Rhapsody Extension
1. Visibility of system status	○	○-
2. Match between system and real world	○	○
3. User control and freedom	○-	○-
4. Consistency and standards	○-	○-
5. Error prevention	+	+
6. Recognition rather than recall	+	+
7. Flexibility and efficiency of use	+	+
8. Aesthetic and minimalistic design	○?	○?
9. Error recognition and recovery	+	+

Table 5.1: Overview of the evaluation.

could not apply them to the model tree, and to some types of diagram elements. Most aspects of our implementation that do not satisfy the usability heuristics are due to tool restrictions. If we could have fully implemented the concepts of Chapter 3, the heuristics would be satisfied in all aspects. Therefore, we suggest to address these issues in future work. Furthermore, we could not evaluate the aesthetics of our extension. Thus, we propose to conduct a user study to evaluate this heuristic in future work.

Nevertheless, we conclude that our extensions improve the usability of Rhapsody and Architect when used for SPLE with pure::variants, since they fix the major usability issues identified in Section 3.1, and provide a UI that follows the usability heuristics in most aspects. They especially reduce errors in the variability information of master projects, and increase efficiency by minimizing user's memory load. To measure how much the usability of the UML tools has improved, a user study would be appropriate. However, it is hard to find users that are familiar with pure::variants' UML extensions, since it is a very special application. Hence, future work could be to conduct a user study. In the next chapter, we discuss related work.

6. Related Work

Having presented our work in the previous chapters, we discuss work related to our thesis.

Since we use visualizations in UML diagrams to improve comprehension of UML models that are used for SPLE, we first present related work that also aims to improve the comprehension of UML models. Then, we present tools that can also be used to visualize variability information.

UML Model Comprehension

In our thesis, we propose and implement visualizations that aim to improve model comprehension concerning the relation between the model and the SPL. The works of Otero and Dolado and Cruz-Lemus et al. also aim to improve comprehension of UML models.

To achieve this goal, Otero and Dolado evaluate the comprehension of dynamic modeling in UML designs by conducting two experiments [OD04]. They use the experiments, to evaluate model comprehension with respect to the type of diagram, the application domain, the combination of dynamic diagram types, etc. As result, they obtain conditions, which, when followed, result in effective comprehension of UML dynamic models

Cruz-Lemus et al. also conduct user studies to evaluate model comprehension [CLGM+09]. However, they concentrate on evaluating the comprehension of statecharts with composite states. They conclude that use of statecharts with composite states may influence model comprehension negatively.

Similar to our thesis, Otero and Dolado and Cruz-Lemus et al. aim to improve the comprehension of UML models. However, they concentrate on evaluating how the usage of UML diagrams and model elements influences model comprehension, whereas

we focus on improving comprehension of UML models related to SPLE by using visualizations. Nevertheless, the results of their work could be used to additionally improve model comprehension when using our extensions.

Visualizing Variability Information

Heidenreich et al. have implemented an extension to Eclipse IDE that supports MDPLE in a similar way as pure::variants' connections to Rhapsody and Architect [HKW08][HSW08]. It is called *FeatureMapper*[1]. Other than pure::variants' UML extensions, it enables users to produce models based on modeling languages of the Eclipse Modeling Framework, such as UML. To these models, features can be mapped. Thus, when using FeatureMapper, it is not necessary to add variability information to a model. Instead, it stores the variability information in an individual mapping file. Based on this mapping and a variant model, users can derive a final project.

To support users in modeling a master project, Feature Mapper provides several visualizations: Similar to our extensions, users can visualize variants by lowlighting elements that would be excluded. Additional visualizations include the *realization view*, which lowlights all elements that are not mapped to the selected feature, and the *context view*, which enables users to assign colors to features, and highlights all mapped elements in the assigned color. We especially consider the *context view* as useful, since it enables users to identify all elements mapped to a feature at first glance.

In summary, FeatureMapper provides similar functionality as pure::variants' connection to Architect and Rhapsody used with our extensions. The major difference is the supported UML tool. FeatureMapper supports all Eclipse-based modeling languages, and thus, is very flexible, whereas we extend Rhapsody and Architect, which are powerful UML tools enabling users to exploit the whole scope of UML concepts (Eichelberger et al. [EES09]). Both can be of advantage. Furthermore, FeatureMapper provides other visualizations than we do, which we can take as inspiration for future work.

Similar to FeatureMapper's *context view*, coloring of features is done in *CIDE (Colored Integrated Development Environment)*[2] by Kästner et al., which is an SPLE tool that processes the source code of a project (Kästner et al. [KTA08]). It processes preprocessor statements, which contain the project's variability information, and provides visualizations based on these statements. Similar to FeatureMapper's context view users can assign colors to features. Based on these assignments, the background of the source code is highlighted, such that developers can identify elements related to a highlighted feature, at one glance.

Feigenspan et al. evaluate the usage of background colors and further develop it in a tool called *Feature Commander*[3] [FSP+]. They show that using background colors in the proposed way can improve program comprehension and user satisfaction. Since UML

[1] http://featuremapper.org/
[2] http://fosd.de/cide
[3] http://www.fosd.de/fc

model elements can be compared to lines of code, and since both contain variability information (either in preprocessor statements or UML constraints), we assume that colors can also be used in UML models to improve model comprehension and user satisfaction. To test whether this assumption is true, we suggest to add a similar visualization to our extension.

7. Conclusion

Model-Driven Product Line Engineering (MDPLE) is a powerful software engineering approach that increases efficiency of the development process by combining Software Product Line Engineering (SPLE) and Model-Driven Engineering (MDE). However, it is used only reluctantly. Since good tool support is an important factor for the success of a new software engineering approach, we suggested that improving the tool support for MDPLE is necessary. Only few tools support MDPLE. For example, pure::variants for IBM Rational Rhapsody (Rhapsody) or Enterprise Architect (Architect).

Since pure::variants for Rhapsody and Architect contains usability issues, we defined the following goals for this thesis:

1. A list of usability problems of UML tools used for SPLE with pure::variants
2. Concepts for a UML tool extension that improves the usability of the tool
3. An implementation of the proposed extension for Architect and Rhapsody that improves the tools' usability

By addressing these goals, we intended to improve the tool support for MDPLE, and thus, increase the success of this software engineering approach.

In Chapter 3, we addressed the first two goals. We argued which aspects of both Rhapsody and Architect hold usability issues, and suggested how to improve the usability. We primarily discovered that the produced UML projects often contain unnoticed errors, such as typing errors, and that variability information of the project could only be edited in an inefficient way. Furthermore, an SPLE related overview of the project was missing. Thus, users could only get detailed textual access to variability information, which hinders the search for errors in variability information and the overall understanding of the model.

To fix these issues, we proposed an extension that offers: Visualizations giving a better overview of the project (cf. Section 3.2), and a constraint editor that prevents errors by providing autocompletion, syntax highlighting, and error checking. We also devised a User Interface (UI) for our extension, such that users can efficiently work with it.

In Chapter 4, we addressed the third goal. To this end, we presented our implementation of the proposed extension for Architect and Rhapsody. We described which concepts we could implement and which we could not realize due to tool restrictions. The following tool restrictions have the most influence on our implementation: Most importantly, in both tools it is impossible to apply visualizations to the model tree, to operations, or to attributes. Furthermore, we could not integrate our extension into Rhapsody's UI. Apart from the tool restrictions, we succeeded in the implementation of both extensions. In Chapter 5, we evaluated them regarding usability. We concluded that the implemented extensions indeed improve the tools' usability. For evaluation, we employed usability heuristics by Nielsen [Nie94a].

Hence, we have fulfilled our goals. We have improved the usability of UML modeling tools used for pure::variants, which is a tool for feature-based (software) product line development. Thus, we have improved tool support for MDPLE, which increases the acceptance of the software engineering approach. Next, we discuss how we can continue our work on the UML extension we implemented.

Future Work

First of all, it is necessary to implement the features we proposed in Chapter 3 that we could not realize due to tool restrictions. In the following list, we give an overview of the missing features:

1. Apply visualizations to the model tree
2. Apply visualizations to attributes and operations
3. Apply visualizations to Rhapsody constraints that are contained in a class
4. Apply visualizations correctly in a sequence diagram of Architect
5. Integrate our extension into Rhapsody's UI

Since Architect's and Rhapsody's performance is reduced when using our extension with huge models, we suggest to improve the performance of our extension. Especially for applying a visualization to a model the performance should be improved. Moreover, a progress indicator should be added to the Rhapsody extension, since it is missing in the current version.

Furthermore, we propose to implement a visualization similar to the color visualizations of FeatureMapper, CIDE, and Feature Commander (cf. Chapter 6). This visualization

should enable users to assign colors to features, and thus to indentify related model elements at first glance. When multiple features apply to one model element (e.g., when the pvSCL expression assigned to the element is "Temperature AND WindSpeed"), both colors would have to be visualized in some way. Since this can lead to very complicated visualizations, we suggest to assign colors to pvSCL expressions rather than features, so that colors cannot overlap, which simplifies the visualization.

Apart from continuing the implementation, we suggest to conduct a user study. Thus, a more accurate evaluation of the extension's usability could be done.

Bibliography

[BCT05] A. W. Brown, J. Conallen, and D. Tropeano. Introduction: Models, Modeling and Model-Driven Architecture (MDA). In *Model-Driven Software Development*, pages 1–16. Springer, 2005. (cited on Page 1)

[BD06] Danilo Beuche and Mark Dalgarno. Software Product Line Engineering with Feature Models. *Methods and Tools*, 14(4):9–17, 2006. (cited on Page 5)

[Ber83] Jacques Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983. (cited on Page 18 and 22)

[BMS95] Curt C. Braun, Paul B. Mineb, and N. Clayton Silver. The Influence of Color on Warning Label Perceptions. *International Journal of Industrial Ergonomics*, 15(3):179–187, 1995. (cited on Page 28)

[CAK+05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-Driven Software Product Lines. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 126–127. ACM Press, 2005. (cited on Page 1)

[CLGM+09] José A. Cruz-Lemus, Marcela Genero, M. Esperanza Manso, Sandro Morasca, and Mario Piattini. Assessing the Understandability of UML Statechart Diagrams with Composite States–A Family of Empirical Studies. *Empirical Software Engineering*, 14(6):685–719, 2009. (cited on Page 51)

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on Page 1, 9, 10, and 11)

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an Information Workspace. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology*, pages 181–186. ACM Press, 1991. (cited on Page 16)

[Die07] Stephan Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007. (cited on Page 17)

[EES09] Holger Eichelberger, Yilmaz Eldogan, and Klaus Schmid. A Comprehensive Survey of UML Compliance in Current Modelling Tools. In *SE '09: Proceedings of the German Conference for Software Engineering*, pages 39–50. Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel, 2009. (cited on Page 2 and 52)

[FSP+] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. How to Scale Background Colors to Support Program Comprehension. In *International Conference on Evaluation and Assessment in Software Engineering*. Submitted January 10, 2011. (cited on Page 52)

[Gal02] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc., 2nd edition, 2002. (cited on Page 19)

[Gar09] Janel Garvin. Users' Choice: 2009 Software Development Platforms – A Comprehensive User Satisfaction Survey of Over 1200 Software Developers. Technical report, Evans Data Corporation, 2009. (cited on Page 11)

[Gol10] E. Bruce Goldstein. *Sensation and Perception*. Cengage Learning, 8th edition, 2010. (cited on Page 17)

[GP96] Jill Gerhardt-Powals. Cognitive Engineering Principles for Enhancing Human-Computer Performance. *International Journal of Human-Computer Interaction*, 8(2):189–211, 1996. (cited on Page 16)

[HBC+92] T. Hewett, R. Baecker, S. Card, T. Carey, J. Gasen, M. Mantei, G. Perlman, G. Strong, and W. Verplank. ACM SIGCHI Curricula for Human-Computer Interaction. Technical Report 608920, ACM SIGCHI, 1992. (cited on Page 14)

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Comp. International Conference Software Engineering (ICSE)*, pages 943–944. ACM Press, 2008. (cited on Page 52)

[HSW08] Florian Heidenreich, Ilie Savga, and Christian Wende. On Controlled Visualisations in Software Product Line Engineering. In *ViSPLE '08: Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering*, pages 335–341. Lero Int. Science Centre, 2008. (cited on Page 52)

[HT06] B. Hailpern and P. Tarr. Model-Driven Development: The Good, the Bad, and the Ugly. *IBM Systems Journal*, 45(3):451–461, 2006. (cited on Page 1)

[ISO98] Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) - Part 11: Guidance on Usability. Technical Report 9241-11, International Organization for Standardization, 1998. (cited on Page 15)

[KB09] Charles W. Krueger and Martin Bakal. Systems and Software Product Line Engineering with SysML, UML and the IBM Rational Rhapsody BigLever Gears Bridge. Whitepaper, 2009. (cited on Page 2)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990. (cited on Page 9 and 10)

[KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. ViSPLE '08: Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering. In *Proceedings SPLC Workshop Visualization in Software Product Line Engineering*, pages 303–312. Lero, 2008. (cited on Page 52)

[Lin94] Gitte Lindgaard. *Usability Testing and System Evaluation: A Guide for Designing Useful Computing Systems*. Chapman Hall, 1994. (cited on Page 15)

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., 2007. (cited on Page 1, 2, and 11)

[Lut07] Robyn Lutz. Tool-Support Survey for Product-Line Verification and Validation Techniques. Technical report, Jet Propulsion Laboratory, California Institute of Technology and NASA Ames Research Center, 2007. (cited on Page 11)

[Lut08] Robyn Lutz. An Evaluation Report for Three Product-Line Tools (FORM, pure::variants and Gears). Technical report, Jet Propulsion Laboratory, California Institute of Technology and NASA Ames Research Center, 2008. (cited on Page 11)

[Mat04] Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 127–136. IEEE Computer Society, 2004. (cited on Page 11)

[MBM09] Mira Mezini, Danilo Beuche, and Ana Moreira, editors. *MDPLE '09: Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering*, 2009. (cited on Page 1)

[Mil56] George Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956. (cited on Page 17 and 46)

[MN90] Rolf Molich and Jakob Nielsen. Improving a Human-Computer Dialogue. *Communications of the ACM*, 33(3):338–348, 1990. (cited on Page 16)

[Moo07] Daniel Moody. What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. In *Advances in Information Systems Development*, pages 481–492. Springer US, 2007. (cited on Page 24)

[Nie94a] Jakob Nielsen. Enhancing the Explanatory Power of Usability Heuristics. In *CHI '94: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence*, pages 152–158. ACM Press, 1994. (cited on Page 16, 17, and 56)

[Nie94b] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 1994. (cited on Page 15, 16, 19, 45, and 48)

[NL06] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders, 2006. (cited on Page 15)

[NM90] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human Factors in Computing Systems: Empowering People*, pages 249–256. ACM, 1990. (cited on Page 16)

[OD04] Mari Carmen Otero and José Javier Dolado. Evaluation of the Comprehension of the Dynamic Modeling in UML. *Journal of Information and Software Technology*, 46(1):35–53, 2004. (cited on Page 51)

[PD10] Bernhard Preim and Raimund Dachselt. *Interaktive Systeme – Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung*. Springer, 2010. (cited on Page 29)

[pur] pure-systems GmbH. *pure::variants User's Guide Version 3.0 for pure::variants 3.0*. (cited on Page 12, 13, and 30)

[pur04] pure-systems GmbH. Variant Management with pure::variants. Technical White Paper, 2004. (cited on Page 2, 11, and 12)

[Ric91] John Rice. Display Color Coding: 10 Rules of Thumb. *IEEE Software*, 8(1):86–88, 1991. (cited on Page 48)

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2nd edition, 2004. (cited on Page 1, 6, 7, 8, and 37)

[Sha91] Brian Shackel. Usability - Context, Framework, Definition, Design and Evaluation. In *Human Factors for Informatics Usability*, pages 21–37. Cambridge University Press, 1991. (cited on Page 15)

[Shn97] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction Third Edition*. Addison-Wesley, 1997. (cited on Page 16, 17, and 49)

[Smi78] Alvy Ray Smith. Color Gamut Transform Pairs. In *SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, volume 12, pages 12–19. ACM Press, 1978. (cited on Page 18)

[Ste06] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 481 – 497. ACM Press, 2006. (cited on Page 2)

[TG80] Anne Treisman and Garry Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12(1):97–136, 1980. (cited on Page 17)

[UML10] OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Technical Report formal/2010-05-03, Object Management Group, 2010. (cited on Page 6)

[Vir91] Robert A. Virzi. A Preference Evaluation of Three Dialing Plans for a Residential, Phone-Based, Information Service. In *Proceedings of the Human Factors and Ergonomics Society 35th Annual Meeting*, pages 240–243. Human Factors and Ergonomics Society, 1991. (cited on Page 15)

[War04] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers Inc., 2004. (cited on Page 17, 18, 23, 24, 26, and 28)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 10. Januar 2011