Software Product Line Engineering

Third Year Project

Matei Vasilache

April, 2020

Explain what I’ve done.

Talk about the meta programming part!!

Explain the problem

Project definition

Write code for a lot of variants (460)

The bigger picture

C pre processor vs methods vs how I did it

# Abstract

This project is a deep dive into the new standardised reality of code production. In any environment, efficient use of one’s resources is key. The purpose of this project is to tackle this problem in the context of software development. This is done by describing the design and automated derivation of variants in a Software Product Line using both an easy to understand and a practical example. Software developers can and should concentrate on creating reusable pieces of code that can combine in multiple variants of deliverable code. This project tackles the design stage by looking at trees for easy visualisation of available combinations by describing the relationships between features in an illustration called the feature tree. Moreover, the project takes a look at the complexities of trying to convey the sheer number of these variants to non-technical customers and analyses 2 possible graphical environments for them to interact with. The automated derivation part of this project involves taking a hands on approach in discovering the importance of Meta programming. This turns out to be the brain that translates the requested features into deliverable code by identifying the needed code snippets, combining them and enforcing requirements.

Table of Contents

[1 Abstract 2](#_Toc69843133)

[2 Introduction 4](#_Toc69843134)

[2.1 Background 4](#_Toc69843135)

[2.2 Motivation 4](#_Toc69843136)

[2.3 Project goal 5](#_Toc69843137)

[2.4 Project Plan 5](#_Toc69843138)

[2.5 Report Structure 5](#_Toc69843139)

[2.6 Impact of COVID-19 5](#_Toc69843140)

[3 Concepts 5](#_Toc69843141)

[3.1 Problem space 5](#_Toc69843142)

[3.2 Solution space 6](#_Toc69843143)

[3.3 Software factory 6](#_Toc69843144)

[3.4 Domain engineering 7](#_Toc69843145)

[3.5 Feature model 8](#_Toc69843146)

[3.6 Feature-oriented programming 10](#_Toc69843147)

[3.7 Product Family Engineering 11](#_Toc69843148)

[3.8 Metaprogramming 11](#_Toc69843149)

[4 Technologies 12](#_Toc69843150)

[4.1 Metaprogramming 12](#_Toc69843151)

[4.1.1 C pre-processor 12](#_Toc69843152)

[4.1.2 Bash 12](#_Toc69843153)

[4.1.3 Python 12](#_Toc69843154)

[4.2 Programming 12](#_Toc69843155)

[4.2.1 Java 12](#_Toc69843156)

[4.2.2 C 12](#_Toc69843157)

[5 Implementation 12](#_Toc69843158)

[5.1 Easy example 12](#_Toc69843159)

[5.1.1 Feature Trees 13](#_Toc69843160)

[5.1.2 Plot features into readable trees. 13](#_Toc69843161)

[5.1.3 Complexities 13](#_Toc69843162)

[5.1.4 Deep trees versus intertwined features 13](#_Toc69843163)

[5.1.5 Code hotspots 13](#_Toc69843164)

[5.1.6 Where the code for the feature is written versus where it is used. 13](#_Toc69843165)

[5.1.7 Requirements 13](#_Toc69843166)

[5.1.8 User Interface 13](#_Toc69843167)

[5.2 Practical example 13](#_Toc69843168)

[5.2.1 Feature Trees 13](#_Toc69843169)

[5.2.2 Plot features into readable trees. 13](#_Toc69843170)

[5.2.3 Complexities 13](#_Toc69843171)

[5.2.4 Deep trees versus intertwined features 13](#_Toc69843172)

[5.2.5 Code hotspots 13](#_Toc69843173)

[5.2.6 Where the code for the feature is written versus where it is used. 13](#_Toc69843174)

[5.2.7 Requirements 13](#_Toc69843175)

[5.2.8 User Interface 13](#_Toc69843176)

[6 Issues 13](#_Toc69843177)

[7 Reflection 14](#_Toc69843178)

[8 Conclusion 14](#_Toc69843179)

# Introduction

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be employed.

While early software product line methods at the genesis of the field provided the best software engineering improvement metrics seen in four decades, the latest generation of software product line methods and tools are exhibiting even greater improvements. New generation methods are extending benefits beyond product creation into maintenance and evolution, lowering the overall complexity of product line development, increasing the scalability of product line portfolios, and enabling organizations to make the transition to software product line practice with orders of magnitude less time, cost and effort.

Recently the concepts of software product lines have been extended to cover systems and software engineering holistically. This is reflected by the emergence of industry standard families like ISO 265xx on systems and software engineering practices for product lines

## Background

The background of the field explored in this project lies in the past and is heavily inspired by the industrialisation era when manufacturing companies developed streamlined ways of extracting more value from the available manpower. In the software development perspective, the resources consist of software developer time and the pieces of code they write. As the name suggests, Software Product Line Engineering is heavily inspired by the manufacturing company ethos. Seeing the great success created by the infatuation of factories in such domains, only one logical next step appears. In the quest of efficiency and customizability, the abilities offered by factories and product lines need to be translated in the software development environment.

## Motivation

A quest for efficiency

Much like car companies that create multiple models using a pool of parts,

Although the term "Software Product line Engineering" is becoming more widely known, there is still uncertainty among developers about how it would apply in their own development context

The characteristic that distinguishes software product lines from previous efforts is predictive versus opportunistic software reuse. Rather than put general software components into a library in the hope that opportunities for reuse will arise, software product lines only call for software artefacts to be created when reuse is predicted in one or more products in a well-defined product line.

Recent advances in the software product line field have demonstrated that narrow and strategic application of these concepts can yield order of magnitude improvements in software engineering capability.[citation needed] The result is often a discontinuous jump in competitive business advantage[citation needed], similar to that seen when manufacturers adopt mass production and mass customization paradigms.

## Project goal

Explore the complexities and understand the global move towards recycling.

## Project Plan

## Report Structure

## Impact of COVID-19

At the time of this report write up, the Coronavirus disease is still very much a part of people’s lives and the world seems to still be far from returning to normal. On one hand, I am part of the second generation of students graduating during this pandemic which means I had more time to adapt and accustom myself to working in such an environment. Lockdowns meant that I could no longer be distracted by constant social activities and I actually started enjoying and seeing the benefits of working from home and virtual meetings. On the other hand, the downsides were still felt. On a micro level, the lack of physical access to university facilities was worrying and lockdowns not only reduced social activities, but stopped them altogether. This had an impact on my mental health as there were little opportunities to wind down and feel included. On a macro level, being a final year student, I had to constantly juggle this massive project, the multiple close calls of getting COVID from getting in touch with later confirmed people and trying to cement my future in a much more reticent job market. The last peace had a great impact as this year was filled with uncertainty and discouragement as multiple companies rejected applications as they were not interested in employing graduates anymore.

# Concepts

## Problem space

Problem space is where the needs of your customers reside. It is where you learn more about users and their problems to enable you to determine what your product needs to do. Essentially, it is the foundation upon which the solution space stands.

However, learning about customers’ needs isn’t the simplest of tasks. It is not always likely that you’ll learn what those needs are easily. The process takes patience and time. To be effective, you require skills that can enable you to win the confidence of the customers and their cooperation.

The seeming difficulty involved in this space can make product management teams want to spend less time in it.

## Solution space

The solution space is where products and product representations that are targeted at the customer live. This is where product managers usually love to spend the bulk of their time.

People are usually encouraged to be problem solvers – to provide solutions. Therefore, it is not surprising that the typical product development team is drawn more to building products. And it may not matter if enough time has been spent understanding the problem to create good requirements – that is where failure starts from.

## Software factory

A software factory is a structured collection of related software assets that aids in producing computer software applications or software components according to specific, externally defined end-user requirements through an assembly process.[1] A software factory applies manufacturing techniques and principles to software development to mimic the benefits of traditional manufacturing. Software factories are generally involved with outsourced software creation.

In software engineering and enterprise software architecture, a software factory is a software product line that configures extensive tools, processes, and content using a template based on a schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components.[2]

Since coding requires a software engineer (or the parallel in traditional manufacturing, a skilled craftsman) it is eliminated from the process at the application layer, and the software is created by assembling predefined components instead of using traditional IDEs. Traditional coding is left only for creating new components or services. As with traditional manufacturing, the engineering is left to creation of the components and the requirements gathering for the system. The end result of manufacturing in a software factory is a composite application.

Software factory–based application development addresses the problem of traditional application development where applications are developed and delivered without taking advantage of the knowledge gained and the assets produced from developing similar applications. Many approaches, such as training, documentation, and frameworks, are used to address this problem; however, using these approaches to consistently apply the valuable knowledge previously gained during development of multiple applications can be an inefficient and error-prone process.

Software factories address this problem by encoding proven practices for developing a specific style of application within a package of integrated guidance that is easy for project teams to adopt. Developing applications using a suitable software factory can provide many benefits, such as improved productivity, quality and evolution capability.[1]

Developing applications using a software factory can provide many benefits when compared to conventional software development approaches. These include the following:

Consistency: Software factories can be used to build multiple instances of a software product line (a set of applications sharing similar features and architecture), making it easier to achieve consistency. This simplifies governance and also lowers training and maintenance costs.

Quality: Using a software factory makes it easier for developers to learn and implement proven practices. Because of the integration of reusable code, developers are able to spend more time working on features that are unique to each application, reducing the likelihood of design flaws and code defects. Applications developed using a software factory can also be verified before deployment, ensuring that factory-specific best practices were followed during development.

Productivity: Many application development activities can be streamlined and automated, such as reusing software assets and generating code from abstractions of the application elements and mechanisms.[1]

Value for developers

Developers can use software factories to increase productivity and incur less ramp-up time. This is achieved through creating a high-quality starting point (baseline) for applications which includes code and patterns. This enables projects to begin with a higher level of maturity than traditionally developed applications. Reusable assets, guidance and examples help address common scenarios and challenges and automation of common tasks allows developers to easily apply guidance in consistent ways. Software factories provide a layer of abstraction that hides application complexity and separates concerns, allowing developers to focus on different areas such as business logic, the user interface (UI) or application services without in-depth knowledge of the infrastructure or baseline services. Abstraction of common developer tasks and increased reusability of infrastructure code can help boost productivity and maintainability.[3]

## Domain engineering

Domain engineering, is the entire process of reusing domain knowledge in the production of new software systems. It is a key concept in systematic software reuse and product line engineering. A key idea in systematic software reuse is the domain. Most organizations work in only a few domains. They repeatedly build similar systems within a given domain with variations to meet different customer needs. Rather than building each new system variant from scratch, significant savings may be achieved by reusing portions of previous systems in the domain to build new ones.

The process of identifying domains, bounding them, and discovering commonalities and variabilities among the systems in the domain is called domain analysis. This information is captured in models that are used in the domain implementation phase to create artifacts such as reusable components, a domain-specific language, or application generators that can be used to build new systems in the domain.

In product line engineering as defined by ISO26550:2015, the Domain Engineering is complemented by Application Engineering which takes care of the life cycle of the individual products derived from the product line

Domain engineering is designed to improve the quality of developed software products through reuse of software artifacts.[2] Domain engineering shows that most developed software systems are not new systems but rather variants of other systems within the same field.[3] As a result, through the use of domain engineering, businesses can maximize profits and reduce time-to-market by using the concepts and implementations from prior software systems and applying them to the target system.[2][4] The reduction in cost is evident even during the implementation phase. One study showed that the use of domain-specific languages allowed code size, in both number of methods and number of symbols, to be reduced by over 50%, and the total number of lines of code to be reduced by nearly 75%.[5]

Domain engineering focuses on capturing knowledge gathered during the software engineering process. By developing reusable artifacts, components can be reused in new software systems at low cost and high quality.[6] Because this applies to all phases of the software development cycle, domain engineering also focuses on the three primary phases: analysis, design, and implementation, paralleling application engineering.[7] This produces not only a set of software implementation components relevant to the domain, but also reusable and configurable requirements and designs.[8]

Given the growth of data on the Web and the growth of the Internet of Things, a domain engineering approach is becoming relevant to other disciplines as well.[9] The emergence of deep chains of Web services highlights that the service concept is relative. Web services developed and operated by one organization can be utilized as part of a platform by another organization. As services may be used in different contexts and hence require different configurations, the design of families of services may benefit from a domain engineering approach.

## Feature model

In software development, a feature model is a compact representation of all the products of the Software Product Line (SPL) in terms of "features". Feature models are visually represented by means of feature diagrams. Feature models are widely used during the whole product line development process and are commonly used as input to produce other assets such as documents, architecture definition, or pieces of code.[citation needed]

A SPL is a family of related programs. When the units of program construction are features—increments in program functionality or development—every program in an SPL is identified by a unique and legal combination of features, and vice versa.

A "feature" is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system".[1] The focus of SPL development is on the systematic and efficient creation of similar programs. FODA is an analysis devoted to identification of features in a domain to be covered by a particular SPL.[1]

Model

A feature model is a model that defines features and their dependencies, typically in the form of a feature diagram + left-over (a.k.a. cross-tree) constraints. But also it could be as a table of possible combinations.[citation needed]

Diagram

A feature diagram is a visual notation of a feature model, which is basically an and-or tree. Other extensions exist: cardinalities, feature cloning, feature attributes, discussed below.

Configuration

A feature configuration is a set of features which describes a member of an SPL: the member contains a feature if and only if the feature is in its configuration. A feature configuration is permitted by a feature model if and only if it does not violate constraints imposed by the model.

Feature Tree

A Feature Tree (sometimes also known as a Feature Model or Feature Diagram) is a hierarchical diagram that visually depicts the features of a solution in groups of increasing levels of detail. Feature Trees are great ways to summarize the features that will be included in a solution and how they are related in a simple visual manner. [2]

Feature models were first introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang in 1990.[1] Since then, feature modeling has been widely adopted by the software product line community and a number of extensions have be

Relationships between a parent feature and its child features (or subfeatures) are categorized as:

Mandatory – child feature is required.

Optional – child feature is optional.

Or – at least one of the sub-features must be selected.

Alternative (xor) – one of the sub-features must be selected

In addition to the parental relationships between features, cross-tree constraints are allowed. The most common are:

A requires B – The selection of A in a product implies the selection of B.

A excludes B – A and B cannot be part of the same product.

As an example, the figure below illustrates how feature models can be used to specify and build configurable on-line shopping systems. The software of each application is determined by the features that it provides. The root feature (i.e. E-Shop) identifies the SPL. Every shopping system implements a catalogue, payment modules, security policies and optionally a search tool. E-shops must implement a high or standard security policy (choose one), and can provide different payment modules: bank transfer, credit card or both of them. Additionally, a cross-tree constraint forces shopping systems including the credit card payment module to implement a high security policy.

## Feature-oriented programming

In computer programming, feature-oriented programming (FOP) or feature-oriented software development (FOSD) is a programming paradigm for program generation in software product lines (SPLs) and for incremental development of programs.

FOSD arose out of layer-based designs and levels of abstraction in network protocols and extensible database systems in the late-1980s.[1] A program was a stack of layers. Each layer added functionality to previously composed layers and different compositions of layers produced different programs. Not surprisingly, there was a need for a compact language to express such designs. Elementary algebra fit the bill: each layer was a function (a program transformation) that added new code to an existing program to produce a new program, and a program's design was modeled by an expression, i.e., a composition of transformations (layers). The figure to the left illustrates the stacking of layers i, j, and h (where h is on the bottom and i is on the top). The algebraic notations i(j(h)), i•j•h, and i+j+h have been used to express these designs.

Over time, layers were equated to features, where a feature is an increment in program functionality. The paradigm for program design and generation was recognized to be an outgrowth of relational query optimization, where query evaluation programs were defined as relational algebra expressions, and query optimization was expression optimization.[2] A software product line is a family of programs where each program is defined by a unique composition of features. FOSD has since evolved into the study of feature modularity, tools, analyses, and design techniques to support feature-based program generation.

The second generation of FOSD research was on feature interactions, which originated in telecommunications. Later, the term feature-oriented programming was coined;[3] this work exposed interactions between layers. Interactions require features to be adapted when composed with other features.

A third generation of research focussed on the fact that every program has multiple representations (e.g., source, makefiles, documentation, etc.) and adding a feature to a program should elaborate each of its representations so that all are consistent. Additionally, some of representations could be generated (or derived) from others. In the sections below, the mathematics of the three most recent generations of FOSD, namely GenVoca,[1] AHEAD,[4] and FOMDD[5][6] are described, and links to product lines that have been developed using FOSD tools are provided. Also, four additional results that apply to all generations of FOSD are: FOSD metamodels, FOSD program cubes, and FOSD feature interactions.

## Product Family Engineering

## Metaprogramming

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running.[1][2] In some cases, this allows programmers to minimize the number of lines of code to express a solution, in turn reducing development time.[3] It also allows programs greater flexibility to efficiently handle new situations without recompilation.

Metaprogramming can be used to move computations from run-time to compile-time, to generate code using compile time computations, and to enable self-modifying code. The ability of a programming language to be its own metalanguage is called reflection.[4] Reflection is a valuable language feature to facilitate metaprogramming.

Metaprogramming was popular in the 1970s and 1980s using list processing languages such as LISP. LISP hardware machines were popular in the 1980s and enabled applications that could process code. They were frequently used for artificial intelligence applications.

Some argue that there is a sharp learning curve to make complete use of metaprogramming features.[9] Since metaprogramming gives more flexibility and configurability at runtime, misuse or incorrect use of the metaprogramming can result in unwarranted and unexpected errors that can be extremely difficult to debug to an average developer. It can introduce risks in the system and make it more vulnerable if not used with care. Some of the common problems which can occur due to wrong use of metaprogramming are inability of the compiler to identify missing configuration parameters, invalid or incorrect data can result in unknown exception or different results.[10] Due to this, some believe[9] that only high-skilled developers should work on developing features which exercise metaprogramming in a language or platform and average developers must learn how to use these features as part of convention.

# Technologies

## Feature Tree Visualization

### Graphviz

### Hand drawing

### Diagram programs

## Metaprogramming

### C pre-processor

### Bash

### Python

#### Tkinter

#### PIL

## Programming

### Java

### C

#### OpenGL

# Implementation

We are finally in a stage where we can take practical steps to understand what SPLE is and how it’s used in reality. All the technical background is explained and it’s time to put it in practice. During this year, I had to constantly move in steps to more and more complicated examples of feature trees to be able to understand and overcome any complexities associated to each step. This project will focus on talking about 2 main examples. The first explained example was my second iteration of progress and the second example equates to my third and last iteration.

## Easy example

This example is a variation of a very easy to understand and solve “hello world” system. This is considered an easy example because of the very linear order of features and made up requirements and this will be showed in the way my metaprogramming will deal with creating the final outcome. Because of the very linear nature, the metaprogramming script is able to simply run each feature when it is its turn without any customization or outside input. In reality, we know that code is more complex than that.

### Feature Trees

### Plot features into readable trees.

### Complexities

### Deep trees versus intertwined features

### Code hotspots

### Where the code for the feature is written versus where it is used.

### Requirements

### User Interface

User to be able to choose variants

## Practical example

Because of the graphical nature of the code and repetitive nature of the features in this game like environment.

Complexity is showed by the intertwined nature of code and large amount of requirements.

### Feature Trees

### Plot features into readable trees.

### Complexities

### Deep trees versus intertwined features

### Code hotspots

### Where the code for the feature is written versus where it is used.

### Requirements

### User Interface

# Issues

So far we have highlighted some of the most common issues that will be encountered when working with Software Product Lines. In this section we highlight some additional issues. Even with visualization support from specialist tools such as pure::variants the visual representation of very complex model structures is a not a completely solved problem.

Larger feature models can have several hundred features and the solution space can have several thousands or more constituents. Thus it can be hard to understand the implications of modifications to these models just through use of model diagrams. Issues around Product Line evolution are also very important. Evolution must be managed since changes that positively affect one or more variants could have a negative effect on other variants and these issues may only show up when variants are produced long after the changes have been implemented. Finally, testing a Product Line also represents a significant challenge. Most Product Lines offer more potential variability than is in use at any one time, and testing all possible variants is usually impossible and in some cases a waste of time. Testing just those variants that are produced is already a difficult problem where there is a high number of variants. However, it is still necessary to co-ordinate testing with variant production in some way. One approach is to create test asset variants as one does for the (software) solution space variants – effectively creating a parallel test solution space that is driven from the Feature model. Reduction of test effort is still an open issue though for many Product Lines.

# Reflection

# Conclusion

The example presented in this article shows how the variability of the problem space of a Product Line can be described very simply using feature models. The automated production of solution variants is the logical next step, for which we have shown one example. In this example we have highlighted some of the approaches and issues that need to be considered when using Software Product Lines. These are covered in more depth in [Bos00], a standard work on Software Product Lines, for example. The authors also welcome comments and questions on any aspect of Software Product Lines Engineering.

These types of problems typically occur in portal or embedded applications, e.g. vehicle control applications [Ste04]. Software Product Line Engineering (SPLE) offers a solution to these not quite new, but increasingly challenging, problems [Cle01]. The basis of SPLE is the explicit modelling of what is common and what differs between product variants. Feature Models [Kan90], [Cza00] are frequently used for this. SPLE also includes the design and management of a variable software architecture and its constituent (software) components. This article describes how this is done in practice, using the example of a Product Line of meteorological data systems. Using this example we will show how a Product Line is designed, and how product variants can be derived automatically. Software Product Lines However, before we introduce the example, we'll take a small detour into the basis of SPLE. The main difference from “normal”, one-of-a-kind software development, is a logical separation between the development of core, reusable software assets (the platform), and actual applications. During application development, platform software is selected and configured to meet the specific needs of the application. The Product Line's commonalities and variabilities are described in the Problem Space. This reflects the desired range of applications (“product variants”) in the Product Line (the “domain”) and their inter-dependencies. So, when producing a product variant, the application developer uses the problem space definition to describe the desired combination of problem variabilities to implement the product variant. An associated Solution Space describes the constituent assets of the Product Line (the “platform”) and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown in Figure 1. Figure 1: Overview of SPLE activities Several different options are available for modelling the information in these four quadrants. The problem space can be described e.g. with Feature Models, or with a Domain Specific Language (DSL). There are also a number of different options for modelling the solution space, for example component libraries, DSL compilers, generative programs and also configuration files. [ Cza00 ]. In the rest of this article we will consider each of these quadrants in turn, beginning with Domain Engineering activities. We'll first look at modelling the problem space - what is common to, and what differs between, the different product variants. Then we'll consider one possible approach for realising product variants in the solution space using C++ as an example. Finally we'll look at how Application Engineering is performed by using the problem and solution space models to create a product variant. In reality, this linear flow is rarely found in practice. Product Lines usually evolve continuously, even after the first product variants have been defined and delivered to customers. Our example Product Line will contain different products for entry and display of meteorological data on a PC. An initial brainstorming session has led to a set of possible differences (variation points) between possible products: meteorological data can come from different sensors attached to the PC, fetched from appropriate Internet services or generated directly by the product for demonstration and test purposes. Data can be output directly from the application, distributed as HTML or XML through an integrated Web server or regularly written to file on a fixed disk. The measurements to make can also vary: temperature, air pressure , wind velocity and humidity could all be of interest. Finally the units of measure could also vary (degrees Celsius vs. Fahrenheit, hPa vs. mmHg, m / s vs. Beaufort). Modelling the Problem Space We will now convert the informal, natural-language specification of variability noted above into a formal model, in order to be able to process it. Specifically, we will use a Feature Model. Feature models are simple, hierarchical models that capture the commonality and variability of a Product Line. Each relevant characteristic of the problem space becomes a feature in the model. A definition of the term "feature" is given in Definition 1. Feature models have a tree structure, with features forming nodes of the tree. Feature variability is represented by the arcs and groupings of features. There are four different types of feature groups: “mandatory", “optional", "alternative" and “or”. When specifying which features are to be included in a variant the following rules apply: If a parent feature is contained in a variant, ● all its mandatory child features must be also contained ("n from n"), ● any number of optional features can be included ("m from n, 0 < = m<=n"), ● exactly one feature must be selected from a group of alternative features ("1 from n"), ● at least one feature must be selected from a group of or features ("m from n, m>1"). Unfortunately, no single standard has yet been agreed for the graphical notation of feature models. However, in the literature, the graphical notation of the original FODA method [Ste04] is common. However this is representable with standard text tools and graph libraries only with difficulty. Therefore in this article a simplified notation has been used. Alternatives and groups of or features are represented with traverses between the matching features. In this representation both colour and box connector are used independently to indicate the type of group. Our notation is shown in Figure 2. Features are an abstract concept for describing commonalities and variabilities. What this means precisely needs to be decided for each Product Line. A feature in this sense is a characteristic of a system relevant for some Stakeholder. Depending on the interest of the Stakeholders a feature can be for the example a requirement, a technical function or function group or a nonfunctional (quality) characteristic. Definition 1: Features Using this notation, our example feature model, with some modifications, is shown in Figure 3: Each Feature Model has a root feature. Beneath this are three mandatory features – "Measurements", "Data Source" and "Output Format". Mandatory features will always be included in a product variant if their parent feature is included in the product variant. Mandatory features are not variable in the true sense, but serve to structure or document their parent feature in some way. Our example also has alternative features, e.g. "External Sensors", "Demo" and "Internet" for data sources. All product variants must contain one and only one of these alternatives. At this stage we can already see one advantage that feature modelling has over a natural-language representation - it removes ambiguities - e.g. whether an individual variant is able to process data from more than one source. When taking measurements any combination of measurements is meaningful and at least one measurement source is necessary for a sensible weather station, to model this we use a group of Or. Usually simple optional features are used, such as the example of the freezing point alarm. Further improvements can also be made by refining the model hierarchy. So the strict choice between Web Server output formats - HTML or XML – can be made explicit. Feature models also support transverse relationships, such as requires and mutually exclusive, in order to model additional dependencies between features other than those already described. So, in the example model, a selection of the “Freeze Point” alarm feature is only meaningful in connection with the temperature measurement capability. This can be modelled by an "Freeze Point" requires "Temperature" relationship (not shown in the figure). However, such relations should be used sparingly. The more transverse relations there are, the harder it is for a human user to visualize connections in the model. When creating a feature model it can be difficult to decide exactly how problem space variabilities are to be represented in the model. In this case it is best to discuss this further with the customer. It is usually better to base these discussions around the feature model, since such models are easier for the customer to understand than textual documents and / or UML models. Formalising customer requirements in this way offers significant advantages later in Product Line development, since many architectural and implementation decisions can be made on the basis of the variabilities captured in the feature model. In the example, the use of the output format XML and HTML can be clarified. The model explicitly defines that the choice of output format is only relevant for Web Server, a format selection is not possible for File or Text output. However, in the context of a discussion of the feature model it could be decided that HTML is also desirable for the on-screen (Window) representation and could also be applicable for file storage. This results in the modified feature model shown in Figure 4. Figure 3: Feature model for our meteorological Product Line Figure 2: Structure and notation of feature models We have added “Plaintext” to the existing features; this was implicitly assumed for output to the screen or to a file. We have modelled the mutual exclusion of XML and screen display (“Text”) using a (transverse) relationship between these features (not shown). The previous discussion describes the basic feature model approach commonly found in the literature, but a number of people have extended this basic approach. To complement the so-called hard relations between features (requires and conflicts) the weakened forms recommends and discourages have been added to many feature model dialects. pure::variants [pure], the variant management tool used for modelling the example in this article, also supports the association of named attributes with features. This allows numeric values or enumerated values to be conveniently associated with features e.g. the wind force required to activate the storm alarm could be represented as a "Threshold" attribute of the feature "Storm Alert". An important and difficult issue in the creation of feature models is deciding which problem space features to represent. In the example model it is not possible to make a choice from the available hardware sensor types (e.g. use of a PR1003 or a PR2005 sensor for pressure). So, when specifying a variant, the user does not have direct influence on the selection of sensor types. These are determined when modelling the solution space. If the choice of different sensor types for measuring pressure is a major criterion for the customer / users, then appropriate options would have to be included in the feature model. This means that the features in the problem space are not a 1:1-illustration of the possibilities in the solution space, but only represent the (variable) characteristics relevant for the users of the Product Line. Feature models are a user-oriented (or marketing-oriented) representation of the problem space, not the solution space. After creating the problem space model we can use it to perform some initial analysis. For example, we can now calculate the upper limit on the number of possible variants in our example Product Line. In this case we have 1,512 variants (the model in Figure 2 only has 612 variants). For such a small number of variants the listing of all possible variants can be meaningful. However, the number of variants is usually too high to make practical use of such an enumeration. Modelling the Solution Space In order to implement the solution space using a suitable variable architecture, we must take account of other factors beyond the variability model of the problem space. These include common characteristics of all variants of the problem space that are not modelled in the feature model, as well as other constraints that limit the solution space. These typically include the programming languages that can be used, the development environment and the application deployment environment(s). Different factors affect the choice of mechanisms to be used for converting from variation points in the solution space. These include the available development tools, the required performance and the available (computing) resources, as well as time and money. For example, use of configuration files can reduce development time for a project, if users can administer their own configurations. In other cases, using preprocessor directives (#ifdef) for conditional compilation can be appropriate, e.g. if smaller program sizes are required. There are many possibilities for implementation of the solution space. Very simple variant-specific model transformations can be made with model-driven software development (MDSD) tools by including information from feature models in the Model-Transformation process. [Voel05] gives an example using the openArchitectureware model transformer. Aspect-oriented programming (AOP) can also be used as a means for the efficient conversion of variabilities in the solution space. Product Lines can also be implemented naturally using "classical" means such as procedural or object-oriented languages. Designing a variable architecture A Product Line architecture will only rarely result directly from the structure of the problem space model. The solution space which can be implemented should support the variability of the problem space, but won't necessarily be a 1:1 correspondence with the architecture. The mapping of variabilities can take place in various ways. Figure 4: Enhanced Feature Model for our meteorological Product Line In the example Product Line we will use a simple object-oriented design concept implemented in C++ . A majority of the variability is then resolved at compile-time or link-time; runtime variability is only used if it is absolutely necessary. Such solutions are frequently used in practice, particularly in embedded systems. The choice of which tools to use for automating the configuration and / or production of a variant plays a substantial role in the design and implementation of the solution space. The range of variability, the complexity of relations between problem space features and solution constituents, the number and frequency of variant production, the size and experience of the development team and many further factors play a role. In simple cases the variant can be produced by hand, but automated tools in the form of Excel and / or small configuration scripts, and also model transformers, code generators or variant management systems will speed production. For modelling and mapping of the solution space variability we use pure::variants [pure] and its integrated model transformation. This uses a Family Model to model the solution space, to associate solution space elements with problem space features, and to support the automatic selection of solution space elements when constructing a product variant. Family models have a hierarchical structure, consisting of logical items of the solution architecture, e.g. components, classes and objects. These logical items can be augmented with information about "real" solution elements such as source code files, in order to enable automatic production of a solution from a valid feature model configuration (more on this later). For each family model element a rule is created to link it to the solution space. For example, the Web Server implementation component is only included if the Web Server feature has been selected from the problem space. To achieve this, a hasFeature('Web Server') rule is attached to the "Web Server" component . Any item below “Web Server” in the Family model can only be included in the solution if the corresponding Web Server feature is selected. A pure::variants screen shot showing part of the solution space is shown in Figure 5. In our example, an architectural variation point arises, among other possibilities, in the area of data output. Each output format can be implemented with an object of a format-specific output class. Thus in the case of HTML output, an object of type HtmlOutput is instantiated, and with XML output, an XmlOutput object. There would also be the possibility here of instantiating an appropriate object at runtime using a Strategy pattern. However, since the feature model designates only the use of alternative output formats, the variability can be resolved at compile-time and a suitable object can be instantiated using code generation for example. In our example solution space a lookup in a text database is used to support multiple natural languages. The choice of which database to use is made at compile-time depending on the desired language. No difference in solution architectures can be detected between two variants that differ only in the target language. Here the variation point is Figure 5: pure::variants screen shot - solution space fragment shown at right embedded in the data level of the implementation. In many cases managing variable solutions only at the architectural level is insufficient. As has already been mentioned above, we must also support variation points at the implementation level, i.e. in our case at the C++ source code level. This is necessary to support automated product derivation. The constituents of a solution on the implementation level, like source code files or configuration files which can be generated, can also be entered in the family model and associated with selection rules. So the existence of the Web Server component in a product variant is denoted using a #define preprocessor directive in a configuration Header file. In addition, an appropriate abstract variation point variable "WEB SERVER" must first be created of the type ps:variable in the family model. The value of this variable is determined by a Value attribute. In our case this value is always 1 if the variable is contained in the product variant. An item of type ps:flagfile can now be assigned to this abstract variable. This item also possesses attributes (file, flag), which are used during the transformation of the model into "real" code. The meaning of the attributes is determined by the transformation selected in the generation step . Here we use the standard pure::variants transformation for C / C++ programs, which produces a C-preprocessor #define- Flags in the file defined by file from these specifications. Separating the logical variation point from the solution makes it very simple to manage changes to the solution space. For example, if the same variation point requires an entry in a Makefile, this could be achieved with the definition of a further source element, of the type ps:makefile, below the variation point "WEB SERVER". Deriving product variants The family model captures both the structure of the solution space with its variation points and the connection of solution and problem space. Not only is the separation of these two spaces important, but also the direction of the connection, since problem space models in most cases are much more stable than solution spaces; the linkage of the solution space to the problem space is more meaningful than the selection of solution items by rules in the problem space. This also increases the potential for reuse, since problem space models can simply be combined with other (new, better, faster) solutions. Now we have all the information needed to create an individual product variant. The first step is to determine a valid selection of characteristics from the feature model. In the case of pure::variants, the user is guided towards a valid and complete feature selection. Once a valid selection is found, the specified feature list as well as the family model serve as input for the production of a variant model. Then, as is described above, the rules of the individual model items are checked. Only items that have their rules satisfied are included in the finished solution. Open Issues in SPLE