Gary J. Chastek (Ed.)

LNCS 2379

Software Product Lines

Second International Conference, SPLC 2
San Diego, CA, USA, August 2002
Proceedings

Springer

Gary J. Chastek (Ed.)

Software Product Lines

Second International Conference, SPLC 2
San Diego, CA, USA, August 19-22, 2002
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Gary J. Chastek
Carnegie Mellon University
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213, USA
E-mail: gjc@sei.cmu.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Software product lines : second international conference ; proceedings /
SPLC 2, San Diego, CA, USA, August 19 - 22, 2002. Gary J. Chastek (ed.). -
Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ;
Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2379)
ISBN 3-540-43985-4

CR Subject Classification (1998): D.2, K.4.3, K.6

ISSN 0302-9743
ISBN 3-540-43985-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg
Printed on acid-free paper SPIN: 10870457 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 2379
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Foreword

Software product lines are emerging as an important new paradigm for software development. Product lines are enabling organizations to achieve impressive time-to-market gains and cost reductions. In 1997, we at the Software Engineering Institute (SEI) launched a Product Line Practice Initiative. Our vision was that product line development would be a low-risk, high-return proposition for the entire software engineering community. It was our hope from the beginning that there would eventually be sufficient interest to hold a conference. The First Software Product Line Conference (SPLC1) was the realization of that hope.

Since SPLC1, we have seen a growing interest in software product lines. Companies are launching their own software product line initiatives, product line technical and business practices are maturing, product line tool vendors are emerging, and books on product lines are being published. Motivated by the enthusiastic response to SPLC1 and the increasing number of software product lines and product line researchers and practitioners, the SEI is proud to sponsor this second conference dedicated to software product lines.

We were gratified by the submissions to SPLC2 from all parts of the globe, from government and commercial organizations. From these submissions we were able to assemble a rich and varied conference program with unique opportunities for software product line novices, experts, and those in between. This collection represents the papers selected from that response and includes research and experience reports.

I would like to take this opportunity to thank the authors of all submitted papers, and the members of the program committee who donated their time and energy to the review process. I offer my special appreciation to Len Bass and Henk Obbink, the program co-chairs, to Gary Chastek, the tireless editor of these proceedings, and to Pennie Walters who assisted in the editing process. We hope you will enjoy the fruits of our labor. Together we are pushing the frontier of software product lines.

August 2002 Linda M. Northrop

Preface

SPLC2 continues to demonstrate the maturation of the field of product lines of software. By their nature, product lines cut across many other areas of software engineering. What we see in the papers presented at this conference is the sharpening of the distinction between software engineering for single system development and software engineering for product lines. The distinction exists not only during the software life cycle (requirements gathering, design, development, and evolution) but also in the business considerations that enter into which systems to build and how to manage the construction of these systems.

We have papers that cover the introduction of product lines and the dynamics of organizations attempting to introduce product lines. We have papers that discuss how to choose which products to produce and how to model the features of those products. All of these topics are essential to the success or failure of a product line within an organization and contribute to the uniqueness of the discipline.

We have several sessions that deal with the discovery, management, and implementation of variability. Variability is perhaps the single most important distinguishing element of product lines as compared to single system development. Identification of variation among products is essential to discover the scope of a set of core development assets and identification of variations within a design is essential to manage the production of products from these core assets.

We also cover specialized topics within normal software engineering and their relationship to product line development. Topics such as necessary tool support, validation of aspects of a system's behavior, and the relationship between product lines and component-based software engineering are also covered within the program.

In short, we have selected a collection of papers that cover a broad spectrum of the areas within product lines of software and are excited about the continued development of the field.

August 2002 Len Bass and Henk Obbink

Organizing Committee

Conference Chair:	Linda M. Northrop (Software Engineering Institute, USA)
Program Co-chair:	Len Bass (Software Engineering Institute, USA) Henk Obbink (Philips, The Netherlands)
Tutorial Chair:	Patrick Donohoe (Software Engineering Institute, USA)
Workshop Chair:	Sholom Cohen (Software Engineering Institute, USA)
Panel Chair:	Paul Clements (Software Engineering Institute, USA)
Demonstration Chair:	Felix Bachmann (Software Engineering Institute, USA)
Proceedings Editor:	Gary Chastek (Software Engineering Institute, USA)

Program Committee

Felix Bachmann (Software Engineering Institute)
Stuart Faulk (University of Oregon)
Frank van der Linden (Philips Medical Systems)
Sergio Bandinelli (European Software Institute)
Cristina Gacek (University of Newcastle)
Nenad Medvidovic (University of Southern California)
Don Batory (University of Texas at Austin)
André van der Hoek (University of California, Irvine)
Michael Moore (NASA/Goddard Space Flight Center)
Joseph H. Bauman (Hewlett Packard)
Jean Jourdan (Thales)
Robert L. Nord (Siemens Research, Inc.)
Günter W. Böckle (Siemens AG)
Peter Knauber (Fraunhofer IESE)
Scott Preece (Motorola)
Jan Bosch (University of Groningen)
Philippe Kruchten (Rational Software, Canada)
Alexander Ran (Nokia Research Center)
Grady H. Campbell (Prosperity Heights Software)
Charles W. Krueger (BigLever Software)
David Sharp (The Boeing Company)
Paul Clements (Software Engineering Institute, USA)
Juha H. T. Kuusela (Nokia Research Center)
Steffen Thiel (Robert Bosch GmbH, Germany)
David M. Weiss (Avaya)

Table of Contents

On the Influence of Variabilities on the Application-Engineering Process of a Product Family*

Lars Geyer and Martin Becker

System Software Research Group
p.o. box 3049
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{geyer, mbecker}@informatik.uni-kl.de

Abstract. Product Families typically comprise a set of software assets, which offer the possibility to configure the product family to the needs of a specific application. The configuration process is driven by the variabilities (i.e., the variable requirements that were implemented into the software assets in the form of variation points). During application engineering, a developer selects a consistent set of variabilities; this set is used to instantiate the family assets to the needed functionality. This paper describes the influence of this configuration step on the application-engineering process of a product family. In addition, it identifies the requirements imposed on a configuration technique by the described product family application-engineering process.

1 Introduction

In many application domains, software products are part of a product line (i.e., systems are used in similar environments to fulfill similar tasks). Experiences in recent years show that it is feasible to take advantage of this relationship by implementing a product line as a product family (i.e., by a common infrastructure), which builds the core of every product. With this basis it is only necessary to configure and adapt the infrastructure to the requirements of the specific application. If product family engineering is done right, this results in a considerable decrease in effort needed for the construction of a single system.

A key problem of the success of a product family is the handling of the differences between family members. The requirements relevant to the product family are typically separated into commonalities and variabilities. The commonalities are easy to handle. Since they are part of every system in the family, they can simply be integrated into the infrastructure and automatically reused during the development of the single applications.

Handling variabilities is more difficult. They are part of only some systems in the family, so, the infrastructure has to define the places in which a variability takes effect. These so-called variation points need to be identified, and there needs to be

* This research was supported by the Deutsche Forschungsgemeinschaft as part of Special Research Project 501.

G. Chastek (Ed.): SPLC2 2002, LNCS 2379, pp. 1–14, 2002.

trace information between the variation points and the corresponding variability. In [4], we proposed a variability model as a model, which captures the variability-related information of a product family, and which, therefore, documents the traceability information to the associated variation points in the assets of the product family.

In this paper, we first present a detailed definition to clarify the meaning of the concepts relevant to the handling of variability and their interconnections. Second, we discuss the handling of the offered variability during the application-engineering process (i.e., we describe how the configuration of the variability in the product family is embedded into the application-engineering process). Third, we analyze the resulting application-engineering process for requirements towards a configuration technique, which is suitable in the context of product families.

In the next section, we define key terms. In Section 3, we describe the embedding of the configuration into the application-engineering process. The subsequent section deals with some specific issues in the context of this embedding. Requirements for a configuration technique are summarized in Section 5. This paper concludes with some closing remarks and a look to the future in Section 6.

2 Concepts and Their Interconnections

2.1 Product Lines and Product Families

At first, we start with the definition of a software product line. A *software product line* or is 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 [1]. A *product family*, on the other hand, is a set of systems built from a common set of software assets [1] (i.e., the systems of a product family are based on the same reuse infrastructure). So, the main difference between these two terms is that they describe two different viewpoints. A product line describes a set of related systems from the viewpoint of common functionality, while the term product family describes a set of systems, which are build on common assets.

The definition of product family uses the term software asset. This term is defined as either a partial solution (e.g., a component or a design document) or knowledge (e.g., a requirements database or test procedures). Engineers use assets to build or modify software products [13]. The reuse infrastructure of a product family comprises a set of assets.

2.2 Commonality and Variability

A *commonality* is a quality or functionality that all the applications in a product family share. As a consequence, commonalities are the elements with the highest reuse potential because an implementation of a commonality is used in all family members. In contrast, a variability represents a capability to change or customize a system [12] (i.e., it abstracts from the varying but coherent qualities in regard to one facet between the members of the family [4]). Variabilities always represent a variable aspect from the viewpoint of the requirements (i.e., they abstract from a variable functionality or a variable quality of the system).

The range of a variability identifies the possible variants that represent the different incarnations of the variability found in the family members. The range may also represent a numerical variability, which offers the possibility to choose the value from a range of integer or real numbers. Other variabilities are simply open issues (i.e., for them, it did not make sense to specify a finite set of variants). Instead, an environment is defined in which a developer has to integrate a specific solution for the variability. In some cases, variants may already exist variants for an open variability, but this set is not finite (i.e., new variants may be integrated for a product family member).

The variability described above deals with qualitative aspects (i.e., different kinds of functionality are chosen from a set of variants). Another type of variability is concerned with quantitative aspects. In some domains, it is necessary to include related functionality with different characteristics more than once. For example, in the domain of building automation systems, a whole set of common and variable functionality describes how a system should control a room. Typically, there are several room types in a building, like offices, computer labs, or meeting rooms. For each room type, the control strategy is the same, but the strategy differs between different room types. This results in a separate specification of the variable functionality for each room type (i.e., a set of variable requirements needs to be instantiated several times during the configuration of the product family). This quantitative type of variability is an orthogonal type compared to the qualitative variabilities; the handling of it is different, as can be seen later in this paper.

2.3 Features

The requirements of a product family are typically abstracted to features. A *feature* is a logical unit of behavior that is specified by a set of functional and quality requirements [5]. They are structured in a feature model [11][8], which creates a hierarchical ordering of the features in a tree. The hierarchy expresses a refinement relationship between features. The high-level functionality groups are typically represented near the root of the tree. These groups are refined on the next step with more detailed features, which are further refined on the next step, and so on.

Features can be classified as mandatory (i.e., they are simply part of a system if its parent in the feature hierarchy is part of the system). Alternatively, features can be classified as optional (i.e., associated with the feature is the decision of whether the feature is part of a family member). In addition, features can be numerical (i.e., associated to the feature is a range of numbers). During the configuration, the value of the feature is chosen from this numeric range.

Related optional features may be arranged in groups that are associated with a multiplicity. In this way, they allow for the definition of dependencies between features, like the classic alternative features in the Feature-Oriented Domain Analysis (FODA) method [11]. Alternative features are represented as a group with an "exactly one"-semantic (i.e., exactly one of the optional features may be part of a system). The "exactly one"-semantic can be weakened to an "at most one"-semantic (i.e., it is permissible to choose none of the alternative features). Furthermore, groups can be used to express mandatory subsets (i.e., in a group of features with an "at least one"-semantic, at least one of the features has to be part of a system).

In addition to these simple dependencies between features, there are more complex relationships. Typically, *requires* and *excludes* relationships are found in the literature

(e.g., in [8]). A requires dependency exists when the integration of a feature requires the integration of another feature. The excludes relationship, on the contrary, exists when two feature as incompatible (i.e., they are not allowed to be part of the same system). But dependencies can also combine more than two features. These relationships are typically formulated in Boolean expressions. Between features with a numeric range, relational or even functional dependencies can be defined. In some cases, it is necessary to hold an equation, which combines the values of numerical features; in this case, we have a relational dependency between the concerned features. In other cases, it is possible to compute the value of one feature out of the values of others, yielding a functional dependency.

A feature model can be used favorably for the description of the commonalities and variabilities of a product family. Typically, the commonalities build the core of the feature tree. Plugged into this core are the variabilities as branches of the feature tree; they are further refined by subordinate features, which can be classified as mandatory or optional. Qualitative variabilities are either represented by optional features or by feature groups. Optional features represent optional variabilities, while feature groups are used if a variability contains more than one variant. Numerical variabilities are expressed by numerical features, and open variabilities are expressed by abstract features (i.e., an abstract feature simply references an environment in which the open functionality has to be integrated). Quantitative variability is expressed by cardinalities associated to the link between a superior feature and a subordinate feature (i.e., a cardinality greater than one allows for a multiple instantiation of the subtree defined by the subordinate feature).

Feature models are a suitable basis for the configuration of the product family (i.e., they can be used as the basic structure of a decision model for the adaptation of the assets of a product family). The feature models can be described easily in a configuration technology, which is used during the configuration process to specify a consistent set of features. This feature set is mapped onto the reuse infrastructure, allowing a fast construction of the core functionality of an application. A detailed description of this process is given in the next section.

2.4 Variation Points

Jacobsen, et al. define a variation point as an identifier of one or more locations in a software asset at which the variation will occur [10]. We restrict this definition to one location, therefore allowing multiple variation points to identify locations where a specific variation will occur (i.e., a variation point always identifies one spot in an asset, but a variability may influence more than one variation point). Nonetheless, variation points reflect the variabilities' impacts in the assets. Typically, there is a set of solutions that can be integrated into one variation point. Each alternative of a variation point implements a solution for a specific variant of the variability.

The solutions, which resolve a variation point, are typically of three types [3]. The first type is a piece of the model in which the variation point is defined (e.g., it may be a piece of code or a part of an architecture). The piece is simply pasted into the model, thereby replacing the variation point. The new piece can contain additional variation points, so the resolution of variation points is a recursive process.

The second type is a variation point representing a parameter whose value is generated. This starts from a simple substitution of a symbol with a value. But it can

also comprise the generation of a piece of the asset's model out of a high-level specification, which is completed by the selection of a variant for a variability. During the generation, new variation points can be inserted into the asset model, also resulting in a recursive resolution of variation points.

The third type simply defines an empty frame in which a solution must be created in a creative task by a developer. This type of variation point always comes into play if the range of the variability is open (i.e., not completely defined during the domain-engineering process). In some cases, a variation point is associated with an open set of alternatives. In order to resolve the variation point, it is possible to either choose one of the alternative solutions or integrate an application-specific solution that is developed in a creative task.

Associated with each variation point is a binding time, at which the resolution of the variation point takes place. Typical binding times are architecture configuration (component selection), component configuration (component parameterization), startup time, and runtime. The detailed dependencies between the resolution of variation points at the specified binding time and the application-engineering process are described in the next section.

The relationship between variabilities and variation points is *n to m* (i.e., one variability may affect several variation points and, conversely, a variation point may be influenced by several variabilities). One variability can have variants, which influence the system not only in different components but also at different binding times. Figure 1 shows the connection between assets, variation points, and variabilities.

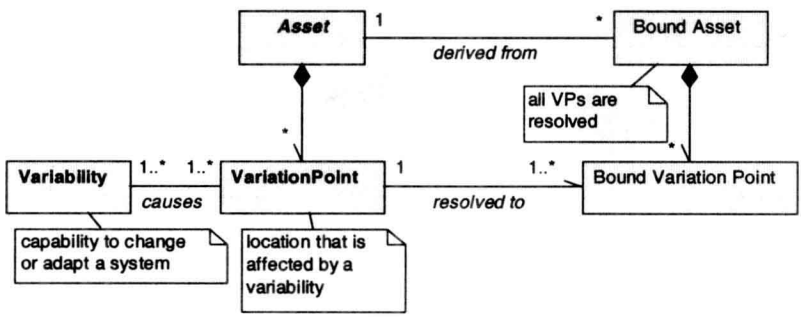

Fig. 1. The Connection Between Variabilities, Variation Points and Assets

3 The Application-Engineering Process

In this section, we will describe the embedding of variability handling into the application-engineering process of a product family. We describe the embedding in this section, and present some integration issues in the next section.

3.1 The Product Family Process

The product family process can be seen in Figure 2. In this figure, the well-known six-pack is presented (i.e., on the upper half of the picture, the domain-engineering process is shown, which comprises the domain analysis, the domain design, and the domain implementation tasks). Each of these steps has one equivalent in the application-engineering process, shown on the lower half of the figure. The counterpart of the domain analysis is the application requirements engineering, followed by the application design and the application implementation task.

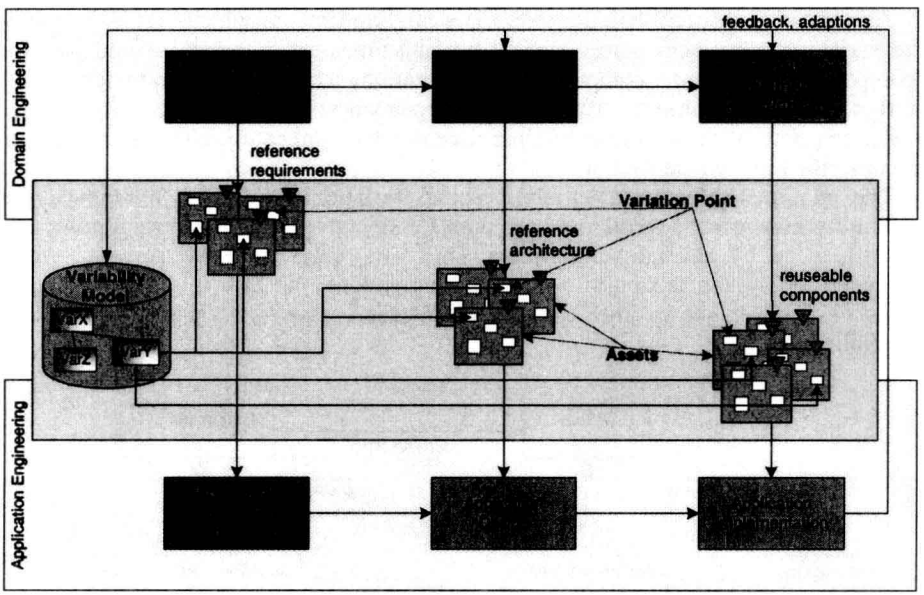

Fig. 2. The Development Process of Product Families: The process comprises a domain-engineering process, which implements the reuse infrastructure shown on asset base between the two subprocesses. The reuse infrastructure is used in the application-engineering process as the core for the development of family members.

In a product family approach, the domain-engineering process is responsible for the creation of reusable assets, which are shown in Figure 2 between the two processes in the middle. The domain-engineering process generates assets on each level (i.e., in the asset base are requirements templates, a generic software architecture, and reusable generic components). Each asset is associated to a specific process task, in which the contained variation points are bound to concrete solutions: the variation points with a binding-time architecture configuration are to be found in assets associated to the application design task. After this task is performed, all variation points with this binding time are bound to a solution. The variation points with a binding-time component configuration are to be found in assets associated to the application implementation task. After performing this task, all variation points with this binding time are bound to a solution.