Software Product Lines in Action

Frank J. van der Linden · Klaus Schmid
Eelco Rommes

Software Product Lines in Action

The Best Industrial Practice in Product Line Engineering

With 90 Figures and 9 Tables

Springer

Frank J. van der Linden
Philips Medical Systems
Veenpluis 4-6
5684 PC Best, The Netherlands
frank.van.der.linden@philips.com

Eelco Rommes
Philips Research
Prof. Holstlaan 4
5656 AA Eindhoven, The Netherlands
eelco.rommes@xs4all.nl

Klaus Schmid
Universität Hildesheim
Institut für Informatik
Samelsonplatz 1
31141 Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Library of Congress Control Number: 2007923180

ACM Computing Classification (2007): D.2, K.6.3, H.4

ISBN 978-3-540-71436-1 Springer Berlin Heidelberg New York

Typesetting: by the authors
Production: Integra Software Services Pvt. Ltd., Puducherry, India
Cover design: KünkelLopka, Heidelberg

Printed on acid-free paper 45/3100/Integra 5 4 3 2 1 0

Foreword

Software product lines represent perhaps the most exciting paradigm shift in software development since the advent of high-level programming languages. Nowhere else in software engineering have we seen such breathtaking improvements in cost, quality, time to market, and developer productivity, often registering in the order-of-magnitude range. Let me say that again: *often registering in the order-of-magnitude range*. Just to be clear, we are talking about software systems built for around one-tenth the cost. With around one-tenth the faults. Delivered in around one-tenth the time. If you know of another way to achieve such a staggering combination of better-faster-and-cheaper, please let me know.

Equally exciting to me are the strategic benefits that accrue to a product line organization, in the form of market agility. At the Software Engineering Institute, we have recorded case study after case study of companies succeeding in one market area with a product line approach, and then taking their production capability to a nearby, under-exploited area of the market, and quickly rising to market dominance in that area as well. And why not? If you can outperform your competitors by order-of-magnitude levels, it's hard to imagine what could keep you from becoming a market leader.

It seems very clear that software product line practice, as a viable and attractive option for software development, is without doubt here to stay. While the underlying concepts are straightforward enough – building a family of related products or systems by planned and careful reuse of a base of generalized software development assets – the devil can be in the details. Successful product line practice can involve organizational change, business process change, and technology change. Bringing comprehensive change to a software development organization isn't easy, and false starts are expensive – or fatal. Better to learn from the experts than strike out through the wilderness on your own.

Which brings us to this book: I fully expect that *Software Product Lines in Action* will become one of the foundational references of this quickly evolving field. It's the most comprehensive treatment of product line practice in

existence today. It's all here, the concepts, a full approach, a holistic treatment of product line practice from the standpoint of business, process, and technology, an analysis method, and a rich collection of case studies. In fact, a big reason to be a fan of this book is its wonderful collection of case studies. Nothing teaches like experience, and the unprecedented ten case studies represent (to my knowledge) the largest collection of experiential software product line reports ever gathered in one reference. More than just third-hand reporting, however, the authors themselves have been integral leaders on many of the case studies on which they report. They have been important contributors to this field almost since it was a field, and can rightly take credit for helping to make software product line practice a known, repeatable, software development approach. In fact, they helped make this field a field. I'm proud to call them colleagues. After you read this book and launch a successful software product line of your own, you'll be proud to call them colleagues too.

Austin, Texas,
January 2007 *Paul Clements*

Preface

The software industry is challenged with a continuous drive to improve its engineering practice. Software has to be produced ever faster and more reliable. Increasingly, complex systems are produced with constant, or even diminishing, numbers of people.

Software product line engineering is a strategic approach to developing software. It impacts business, organisation and technology alike and is a proven way to develop a large range of software products and software-intensive systems fast and at low costs, while at the same time delivering high-quality software.

This book captures the wealth of knowledge that eight companies have gathered during the introduction of the software product line engineering approach in their daily practice.

Who This Book Is For

This book is meant for anyone who is interested in the practical side of product line engineering. Those who consider to use a product line approach in their organisations, those who are about to start one and those who want to improve their current practices will find useful information. This book presents a broad view on product line engineering so that both managers and technical specialists will benefit from reading it. Specific emphasis is given to providing real-world data to support managers in deciding on the potential adoption of product line engineering in their organisations. We believe that best practices are best communicated along with what goes wrong if one fails to adhere to them. This book is also a tool on how to do it right (or wrong), and to learn from the experiences of others.

Background knowledge of product line engineering is not required, but the reader is expected to be familiar with current software engineering practices, or to have some experience in software development.

Readers who want a detailed introduction to the subject are referred to the textbook *Software Product Line Engineering* [106], which describes the foundations, principles and techniques of software product line engineering.

What You Will Learn from Reading This Book

This book gives a practical overview to software product line engineering, driven from industrial experiences that were collected from organisations of varying sizes and domains. Practitioners themselves report on practical implementation: *from practitioners to practitioners*.

This book is complemented with business-related information regarding the benefits and drawbacks of the approach. It not only shows how software product lines can improve the software creation process, but also describes problems that may occur and how companies have solved them in their respective contexts.

The core of this book contains ten case studies, covering small and large organisations, acting in all kinds of domains, with different degrees of domain and process maturity. These companies work on a large variety of software intensive systems including medical imaging, mobile phones, software for televisions, utility control, supervision and management, financial services and car electronics.

The reader will

- understand the relevant aspects, regarding business, architecture, process and organisational issues, of applying software product line engineering.
- learn about the current practice of product line engineering in leading companies of different sizes, operating in several countries and working in various domains.
- have the information for performing an informed analysis on the applicability, or improvement, of the product line approach to his or her own organisation.
- have information about the first steps in transitioning, or improving, the product line approach in his or her organisation.

The Case Studies

Starting in the 1990s, massive investments were made in Europe in the area of product line engineering. This was done both inside companies and as a part of large projects in which companies, research institutes and academia from many different countries co-operated, for example the ESAPS, CAFÉ and FAMILIES projects. One of the results was a flourishing community of product line engineering research and practice in Europe.

The case studies in this book reflect the experiences of companies that were involved in these projects. Each case study was written with experts from the case in question.

The majority of these studies deals with software intensive systems, mainly because the software intensive systems industry in general is more advanced when it comes to software product line engineering. There are several reasons for that.

- Experience with platforms and customisation in other engineering disciplines is often already present in these companies.
- Their customers are used to choosing from a range of systems, each with different properties.
- Pure software customers are often acquainted to and mostly accept the one-size-fits-all system, where they can adapt the system to their specific needs.

However, as more and more information systems — especially business-oriented systems — must be adapted to business-oriented workflows, product line engineering becomes increasingly important. Also for these systems, the adaptation and customisation costs for the client may become prohibitive if systems are delivered with too much diverse and undocumented variability.

The Structure of This Book

This book consist of three parts: a high-level introduction to software product line engineering, ten industrial case studies and their analysis.

Part I — Aspects of Software Product Line Engineering

This part sets a common framework for the description of our industrial case studies. It covers the four major concerns of software product line engineering: Business, Architecture, Process and Organisation. These *BAPO* concerns are a main organising principle of this part. Each of them is explained in detail in a separate chapter. In addition, the Family Evaluation Framework is based on these concerns.

Chapter 1 Product Line Engineering Approach provides the basics you need to understand the book. It explains what product line engineering is, provides an overview of the major aspects (BAPO) and introduces the main topics of software product line engineering: variability and the use of a platform.

Chapter 2 Business explains the business aspects of software product line engineering. It deals with the motivation to initiate or continue with this approach and it explains the economical aspects of software product line engineering.

Chapter 3 Architecture deals with the technical aspects of product lines, most importantly how to deal with variability.

Chapter 4 Process describes the processes for software product line engineering. It describes the separation between domain and application engineering, and the relation between these two life-cycles.

Chapter 5 Organisation deals with roles and responsibilities, structures and distribution of the work.

Chapter 6 The Family Evaluation Framework introduces a BAPO-based framework that can be used to evaluate software product line engineering in larger or smaller parts of companies.

Part II — Experience Reports

This part is the body of the book. It consists of eight experience reports from ten different companies of various sizes and working on various domains.

Chapter 7 Experiences in Product Line Engineering describes the origins of the experiences, the part of software product line engineering that is covered and the formats used within each experiment. The following chapters each describe the company's (or division's) experiences.

Chapter 8 AKVAsmart shows a small company introducing a product-line approach for its range of fish-farming products.

Chapter 9 Bosch Gasoline Systems describes how a product line organisation was set up and executed for a large supplier of automotive products.

Chapter 10 DNV Software deals with the introduction of a product line in ship classification software.

Chapter 11 market maker Software AG shows the business impact of the introduction of a product line on a small company producing financial software.

Chapter 12 Nokia Mobile Phones gives information on the way a product line improves the way to deal with quality requirements, in a large telecom product company.

Chapter 13 Nokia Networks shows another part of this big company, describing the impact of the complex organisation to the product line development.

Chapter 14 Philips Consumer Electronics Software for Televisions shows how all BAPO concerns are affected by the introduction of a product line within this large company

Chapter 15 Philips Medical Systems describes another part of this large company, and how they took a different approach in the introduction of product lines.

Chapter 16 Siemens Medical Solutions shows the difficulties a big company may have to introduce only partially a product line.

Chapter 17 Telvent gives details of the application of an architecture pattern for product lines in the network management domain.

Part III — Conclusions

In this part, conclusions are drawn from an analysis of the set of experiences described. It summarises the lessons learned and provides general guidelines on how to get started with software product line engineering.

Chapter 18 Analysis reflects on the experiences and looks at them from a BAPO and FEF perspective.

Chapter 19 Starting with Software Product Line Engineering presents the steps that need to be taken to successfully make the transition towards software product line engineering, using examples from Part II.

Chapter 20 Outlook looks at trends and expectations for the future. It also describes the challenges that still need to be solved.

Acknowledgements

We thank Eureka/ITEA, BMBF (Germany), SenterNovem (Netherlands) and all other public authorities for funding the projects ESAPS (1999–2001), CAFÉ (2001–2003) and FAMILIES (2003–2005). This book is based on the experience obtained in these projects. However, a lot of additional work was done by a lot of people from the various organisations in order to achieve a quality of the case studies that we can now present here. This goes well beyond project-based work. Each of them is named for the respective case studies, as they were strongly involved and contributed to writing the final case studies.

The BAPO model of product line engineering concerns was a result of the Composable Architectures project conducted at Philips Research between 1998 and 2002. The authors thank their organisations for giving them time to finish the book. These are Philips Medical Systems, Philips Research, Fraunhofer IESE and the University of Hildesheim.

Timo Käkölä provided valuable comments on an earlier version of the book. Monika Lamping did a great job at proof-reading and performing corrections on it and Dennis Stender provided assistance with some pictures. Several figures, viz. 1.1, 1.2, 1.4, 1.5, 2.3, 4.1, 4.2 and 5.1 are copies from the book Software Product Line Engineering [106].

And finally, we would like to thank Ralf Gerstner for accompanying this project for such a long time.

Contents

Part II Experience Reports

Part III Conclusions

Part I

Aspects of Software Product Line Engineering

1

The Product Line Engineering Approach

Software increasingly becomes the key asset for modern, competitive products. No matter how simple or complex, no matter how large or small, there is hardly any modern product without software. Thus, competitiveness in software development has increasingly become a concern for companies of all sizes and in all markets. As a result, product line engineering has gained increasing attention over recent years. While many introductions of a software product line engineering approach are driven by cost and time to market concerns, it supports other business goals as well. We will discuss the consequences of such an approach on business performance in detail in Sect. 1.1. In Sect. 1.2 we will discuss the historical basis of software product line engineering and compare it with other efforts for software reuse. Section 1.3 gives an overview of the most fundamental concepts of product line engineering. These are variability management, business-centric development, architecture-centric development, and the two-life-cycle approach. We will discuss each of these concepts below in detail in a separate section.

1.1 Motivation

Many different reasons lead companies to embark on a software product line engineering approach. These range from more process-oriented aspects like cost and time over product qualities like reliability to end-user aspects like user interface consistency.

The move towards software product line engineering is usually strongly based on economic considerations. Due to its support of large-scale reuse, such an approach improves mostly the process side of software development, i.e. it reduces costs, time to market and improves qualities of the resulting products like their reliability.

The improvement of costs and time to market are strongly correlated in software product line engineering: the approach supports large-scale reuse during software development. As opposed to traditional reuse approaches [108],

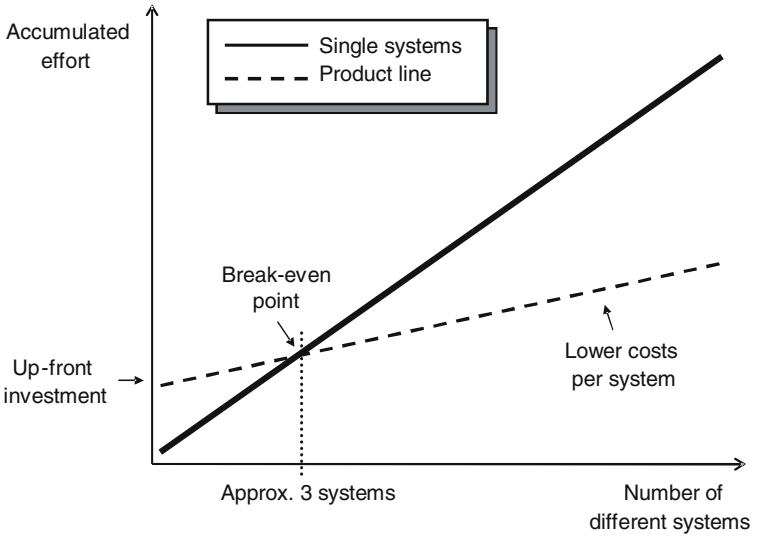

Fig. 1.1. Economics of software product line engineering

this can be as much as 90% of the overall software. Reuse is more cost-effective than development by orders of magnitude. Thus, both *development costs* and *time to market* can be dramatically reduced by product line engineering.

Unfortunately, this improvement does not come for free, but requires some extra up-front investment. This is required for building reusable assets, transforming the organisation, etc. Different strategies exist to make this investment. They range from the so-called *big-bang* approach to an incremental strategy [131], but the underlying need for a set-up investment remains. The positive message is, however, that usually a break-even is reached after about three products, sometimes earlier (cf. Fig. 1.1).

Usually, along with the reduction of development costs a reduction of *maintenance costs* is also achieved. Several aspects contribute to this reduction; most notably the fact that the overall amount of code and documentation that must be maintained is dramatically reduced. As the overall size of the application development projects is strongly reduced, the accompanying *project risk* is reduced as well.[1]

Software product line engineering also has a strong impact on the quality of the resulting software. A new application consists, to a large extent, of matured and proven components. This implies that the defect density of such products can be expected to be drastically lower than products that are

[1] Note that this holds only for individual system development. The overall setup of the product line is actually larger than a single system development and, as a consequence, it is usually more risky. This is often compensated by an incremental build-up strategy for the product line [131]

developed all anew. This leads to more *reliable* and *secure* systems. As a result, *safety* is positively impacted as well. Software product line engineering can also support *quality assurance*, e.g. by regarding a product and its simulation as two variants. Especially for embedded systems, simulation enables extensive testing that would be impossible otherwise. If both variants are derived from the same code, simulations can actually be used as a basis for analysing the quality of the final product. While arguments of costs typically dominate the product line engineering debate, the ability to produce higher quality is for some organisations the major argument, especially in safety-critical domains, where major efforts go in the quality assurance and certification efforts.

Beyond process qualities, software product line engineering positively impacts product aspects like the usability of the final product, e.g. by improving the consistency of the user interface. This is achieved by using the same building blocks for implementing the same kind of user interaction, e.g. by having a single component for installation or user registration for a whole set of products instead of having a specific one for each product. In some cases, demand for this kind of unification has been the basis for the introduction of a product line engineering approach in the first case [20].

1.2 A Brief History of Software Product Line Engineering

The dream of massive software reuse is about as old as software engineering itself. Numerous attempts or initiatives to reuse software were made, but usually with little success. These reuse initiatives were usually based on an approach focusing on small-scale, ad hoc reuse (i.e. typically on the code-level – or at least within a development phase; in addition the development of new assets was rarely based on the systematic analysis of future variability).

The concept of focusing on a specific domain as a basis for developing reusable assets was only introduced somewhat later [97]. However, work in this context focused almost exclusively on the fully automatic development of software in a single domain based on generation tools. This lead to domain-specific languages, but so far never scaled to large-scale system development.[2]

Back in the 1970s Parnas [101] already proposed the concept of *product families*. While it was initially aimed at variability in non-functional characteristics, the product line concept can be traced back to this work.

The concept of product lines was fully introduced in the early 1990s. One of the first contributions was the description of the Feature-Oriented Domain Analysis (FODA) method [80]. Around the same time several companies started to address the issue more systematically. For example, Philips introduced the Building-Block method in the early 1990s [146]. These first

[2] Current developments in domain-specific languages are revisited in Chap. 20, p. 309

approaches were leveraged by massive investments in Europe in the area of software product line engineering, both inside companies and part of several scientific projects. The following are among them:

- ARES (1995–1998) – Architectural Reasoning for Embedded Systems.
- Praise (1998–2000) – Product-line Realisation and Assessment in Industrial Settings.
- ESAPS (1999–2001) – Engineering Software Architectures, Processes and Platforms [49].
- CAFÉ (2001–2003) – from Concepts to Application in system-Family Engineering [35].
- FAMILIES (2003–2005) – FAct-based Maturity through Institutionalisation, Lessons-learned and Involved Exploration of System-family engineering [51].

These projects supported the systematic building of a community of software product line engineering research and practice in Europe. During the same time, especially, the SEI (Software Engineering Institute) supported the development of software product line engineering in the USA, most notably in the context of governmental organisations.

1.3 Fundamentals of the Software Product Line Engineering Approach

The key difference between traditional single system development and software product line engineering is a fundamental shift of focus: from the individual system and project to the product line. This shift especially implies a shift in strategy: from the ad hoc next-contract vision to a strategic view of a field of business.

Software product line engineering relies on a fundamental distinction of development *for reuse* and development *with reuse* as shown in Fig. 1.2. In *domain engineering* (development for reuse) a basis is provided for the actual development of the individual products. As opposed to many traditional reuse approaches that focus on code assets, the product line infrastructure encompasses all assets that are relevant throughout the software development life-cycle. The various assets cover the whole range from the requirements stage over architecture and implementation to testing. This range of assets together defines the product line infrastructure. A key distinction of software product line engineering from other reuse approaches is that the various assets themselves contain explicit variability. For example, a representation of the requirements may contain an explicit description of specific requirements that apply only for a certain subset of the products.

The individual assets in the product line infrastructure are linked together just like assets in software development. For example, traceability is defined

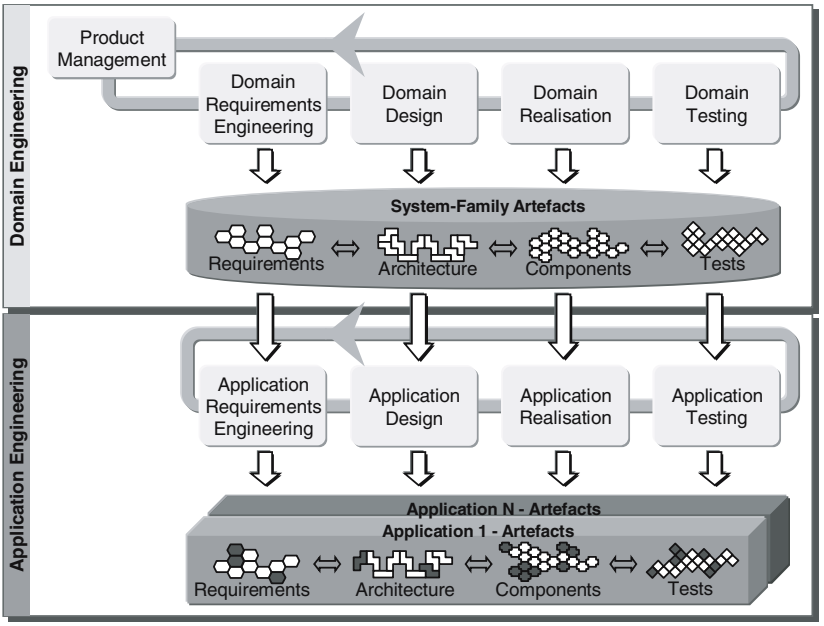

Fig. 1.2. The two-life-cycle model of software product line engineering

among the individual assets, ideally enabling one to take a requirement and identify all related implementation code and test cases.

Application engineering (development with reuse) builds the final products on top of the product line infrastructure. Application engineering is strongly driven by the product line infrastructure, which usually contains most of the functionality required for a new product. The variability explicitly modelled in it provides the basis for deriving the individual products. Basically, when a new product is developed, an accompanying project is set up. Then requirements are gathered and directly categorised as being part of the product line (i.e. a commonality or variability) or product-specific. Then the various assets (e.g. architecture, implementation, etc.) may be instantiated right away, leading to an initial product version. At this stage in the development, up to 90% of the product may be available from reuse; only the remaining 10% must be developed in further steps.

Several principles are fundamental to successful software product line engineering. They can be described as follows:

- *Variability management*: individual systems are considered as variations of a common theme. This variability is made explicit and must be systematically managed.

- *Business-centric*: software product line engineering aims at thoroughly connecting the engineering of the product line with the long-term strategy of the business.
- *Architecture-centric*: the technical side of the software must be developed in a way that allows taking advantage of similarities among the individual systems.
- *Two-life-cycle approach*: the individual systems are developed based on a software platform. These products – as well as the platform – must be engineered and have their individual life-cycles.

In the following sections, we will discuss each of these principles in detail.

1.4 Variability Management

Software product line engineering aims at supporting a range of products. These products may support different, individual customers or may address entirely different market segments. As a result, variability is a key concept in any such approach. Instead of understanding each individual system all by itself, software product line engineering looks at the product line as a whole and the variation among the individual systems. This variability must be defined, represented, exploited, implemented, evolved, etc. – in one word *managed* – throughout software product line engineering. This is what we mean, when we discuss *variability management*.

1.4.1 Types of Variability

When managing variability in a product line, we need to distinguish three main types:

1. *Commonality*: a characteristic (functionality or non-functional) can be common to all products in the product line. We call this a commonality. This is then implemented as part of the platform.
2. *Variability*: a characteristic may be common to some products, but not to all. It must then be explicitly modelled as a possible variability and must be implemented in a way that allows having it in selected products only.
3. *Product-specific*: a characteristic may be part of only one product – at least for the foreseeable future. Such specialties are often not required by the market per se, but are due to the concerns of individual customers. While these variabilities will not be integrated into the platform, the platform must be able to support them.

During the life-cycle of the product line, a specific variability may change in type. For example, a product-specific characteristic may become a variability. On the other hand, a commonality may become a variability as well – for example, if over time the decision is made to support alternatives to the

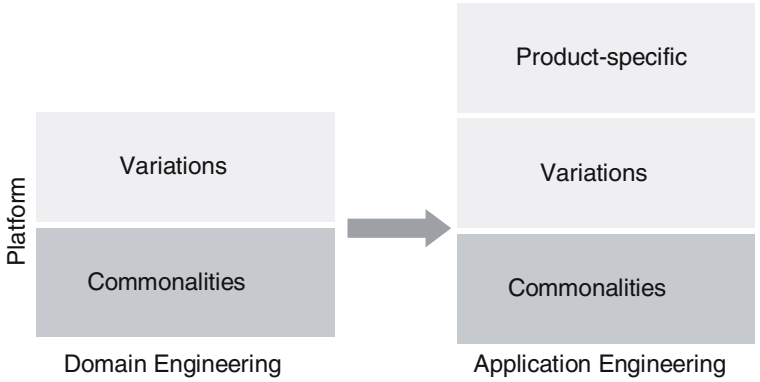

Fig. 1.3. The relation of different types of variability

characteristic (e.g. extending the platform beyond the initial operating system which provided the starting basis).

While commonalities and variabilities are handled mostly in domain engineering, product-specific parts are handled exclusively in application engineering. This is shown in Fig. 1.3.

Variability management is a concern in any software product line engineering approach. It covers the whole life-cycle. It starts with the early steps of scoping, covering all the way to implementation and testing and finally going into evolution. As such, variability is relevant to all assets throughout software development. It is thus a very generic question: How to represent variability?

1.4.2 Variability Representation

Many different approaches to variability representation have been discussed over the years. They differ in several dimensions. The following are among them:

- Which concepts are used to characterise variability?
- Is variability representation integrated with the final assets?

For representing variability several different approaches have been devised. While most modern approaches use features as basic concepts for variability representation, other approaches exist as well. For example, some approaches make the underlying decision that differentiates among various products the basic concept of variability representation. Also, various different interpretations of the term *feature* exist. This opens a very wide spectrum. The major point is that characteristics of the products that differentiate them from other products are core to the representation. Most modern approaches support the characterisation of variability by means of characteristics that cut across different views. This brings us to the second dimension. This dimension describes

whether the variability information is fully integrated in other models or not. While initially variability modelling was often integrated in the underlying notation, meanwhile it is generally recognised that approaches to variability modelling that rely on the distinction of a variability model and a main system model are much easier to apply in complex settings and scale much better [10]. The notion of distinguishing between variability model and basic system model is also called orthogonal variability modelling [106]. There are different approaches to describe the variability model: the decision-based modelling approach relies on describing those decisions that must be made in order to derive a specific product line instance. It was initially described in [135], more recent approaches include [96, 128].

Alternatively, the variability can be described based on a graphical notation. For the sake of simplicity, we will use the same graphical notation here that was used in [106], which goes back to [10]. The notational elements of these variability diagrams are described in Fig. 1.4.

These elements have the following meaning:

- *Variation point*: the variation point describes where differences exist in the final systems (e.g. systems may differ with respect to the operating systems they rely on, with respect to whether they support e-mail or not, etc.).

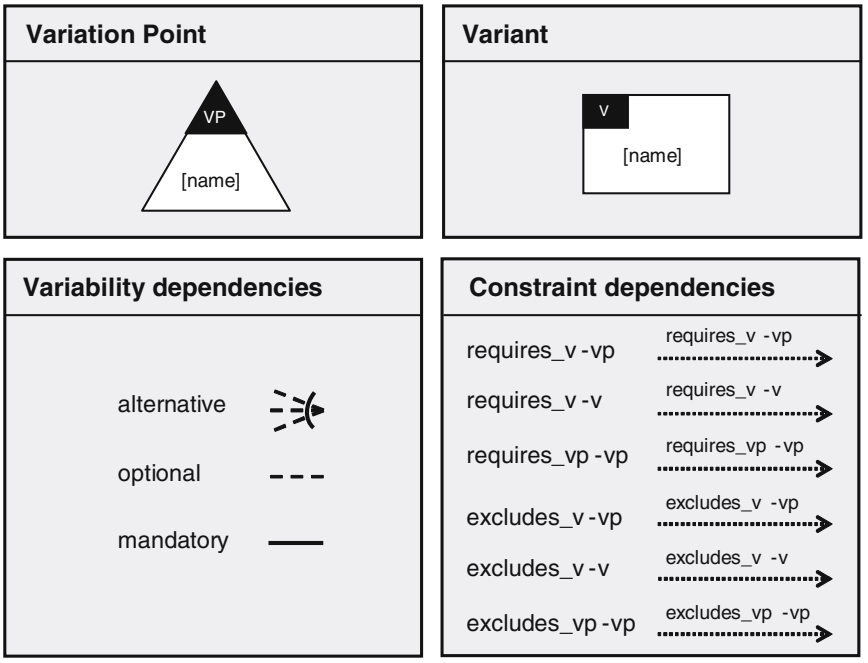

Fig. 1.4. Graphical notation for variability models

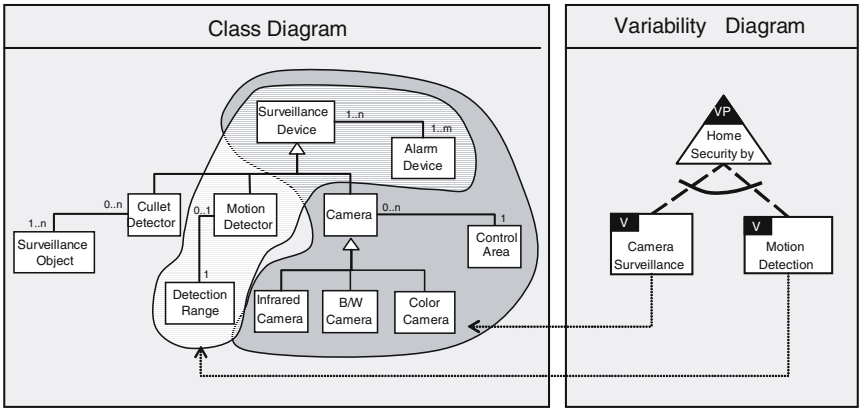

Fig. 1.5. Relation between variability model and class model

- *Variant*: the different possibilities that exist to satisfy a variation point are called variants.
- *Variability dependencies*: this is used as a basis to denote the different choices (variants) that are possible to fill a variation point. The notation includes a cardinality which determines how many variants can be selected simultaneously (e.g. a program may support e-mail, fax, phone, etc. as communication modes even in a single system).
- *Constraint dependencies*: they describe dependencies among certain variant selections. There are two forms:
  1. *Requires*: the selection of a specific variant may require the selection of another variant (perhaps for a different variation point).
  2. *Excludes*: the selection of a specific variant may prohibit the selection of another variant (perhaps for a different variation point).

The variability model on its own is not able to represent the full meaning of variability in software product line engineering. In addition, we need the traditional views on requirements, design, etc. and the relation between these views and the variability, so that we know how variability will have an impact on these individual views. An example of this relationship is given for the case of a UML class model in Fig. 1.5.

1.4.3 Application Engineering and Variability

Setting up the product line infrastructure is not a goal in itself. The ultimate aim is its exploitation during application engineering. This is also called the *instantiation* of the variability.

As new requirements are captured during application engineering, each requirement must be considered and a decision must be made about its future

treatment in the life-cycle: shall it be part of the platform and shall it be a variable part there – or shall it be delegated to product development?

The simplest case is when the product line infrastructure already supports the requirement. In this case, it only needs to be checked that the *binding time* of the infrastructure also supports the requirements. The binding time describes when the decision upon selection of a variant must be made. Typical values are compile-time, link time, start-up time, etc.

If the requirement is not supported by the product line infrastructure, there are three different possibilities:

1. One can try to negotiate to drop or replace it. While this may sound strange from a customer-satisfaction perspective, it needs to be evaluated nevertheless. In a product line context, the more variabilities we must support, the more difficult the evolution of the infrastructure gets.
2. The new requirement shall be integrated with the product line infrastructure. There exist systematic approaches to make such a decision: the so-called *scoping* approaches [119] (see Sects. 1.5 and 2.4).
3. The new requirement shall be integrated in an application-specific manner.

The second case leads to a hand-over with domain engineering, while the third case leads to stand-alone software development in application engineering. Typically, all three cases occur for different requirements in the same system development.

1.5 Business-Centric

While traditional software development focuses on the individual system, product line engineering must always address the market as a whole. Product line engineering can only be successful if the product line infrastructure is in the long term an adequate instrument to field new products onto the market very efficiently. As a consequence, development decisions for the individual product are always linked to the product line at large. This relationship must be managed from an economic point of view.

Because of this strong linkage, it is of key importance that the major business goals for the product line initiative are well understood. Typical business goals are effort- (and thus cost-) reduction, as well as time-to-market reduction. Another major set of goals are quality-related. Typical examples are reliability improvement or usability improvement. The goal of usability improvement is supported as product line engineering inherently supports user interface consistency.

The specific set of goals that provides the basis of a product line engineering effort influences decisions about when a requirement should be implemented and whether it should be implemented for the product line as a whole or only for a specific product. As a rule of thumb, the break-even from

a cost-point of view is typically about three implementations.[3] When three or more product realisations of a requirement are required, it is usually more cost-effective to implement it once as part of domain engineering. This break-even point shifts as soon as additional goals come in like consistency of user interfaces – in this case a single product-specific implementation may already violate this goal.

A business-centric approach to product line engineering entails that key decisions about which functionality to include in the product line and how this support shall be realised (as part of domain or application engineering) is based on a systematic economic decision. This analysis is also called *scoping*. We can differentiate three major categories [119]:

1. *Product portfolio planning*: this aims at determining the specific products and their functionalities that shall be supported by the product line infrastructure.
2. *Domain potential analysis*: this aims at analysing the potential of the product line domain or specific sub-domains in order to identify whether a promising case for product line engineering exists.
3. *Asset scoping*: this determines which specific components shall be built in domain engineering and which requirements they shall support.

Product portfolio planning aims at capturing the products that shall be part of the product line and identifying their main requirements. At this stage, a first overview of commonalities and variabilities of the products is gained. Product portfolio planning is the first step at which an optimisation can (and should) occur. This activity is mostly performed from a marketing point of view [82], but in the context of product line engineering, technical aspects must be taken into account as they strongly impact the production cost [23].

Domain potential analysis focuses on the systematic analysis of an area of functionality in order to determine whether an investment in software product line engineering shall be made. This is sometimes done on the level of the product line as a whole [12], while other approaches focus on individual areas within the product line [122]. The key issue of this step is always to get a systematic answer to the question where reuse investments should be focused. The overall result of this activity corresponds basically to an assessment.

Finally, asset scoping aims at defining the individual components that shall be built for reuse. In order to adequately define these components, two viewpoints must be brought together. These are the viewpoints of business and of architecture. Thus, this activity can be considered as being on the borderline between business-centric and architecture-centric. The business-centric viewpoint can be addressed by an economic analysis [122], while the architectural viewpoint is usually taken into account by an architectural review.

[3] More detailed analysis [123] shows that the break-even point strongly depends on the specific situation. It may range from just above 1 to almost a factor of 10. The underlying driver to this variation is mostly the overhead complexity incurred by the developed genericity